REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Oct95	SBIR Final Report, Ph1, Oct-Mar 95

4. TITLE AND SUBTITLE

Object-Oriented Visual Programming Language
SBIR Final Report, Phase 1

5. FUNDING NUMBERS

Contract Number
DAAH01-95-C-R101

6. AUTHOR(S)

Douglas J. Davenport, PI

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

SNAP Technologies, Inc.
24 Upper Creek Road
POB 0247
Etna, NY]3062-0247

8. PERFORMING ORGANIZATION REPORT NUMBER

VOOPEFR1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Missile Command
AMSMI-RD-PC-GX, Bldg 5400, Rm B145
Redstone Arsenal, AL 35898

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

~~No restrictions~~

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A general purpose visual programming language, based on object-oriented methodology, offers substantial benefits of reusable code, shorter creation and maintenance time, faster overall understanding of the code, and reduced staff learning time. It should be an ideal fit for sites with long-lived code or very large bodies of code. The value of visual programming languages is well understood for very specific tasks; this report reviews the plan for research and design of a general purpose visual object-oriented programming language through nine steps, which includes general specifications for the preferred language and environment.

14. SUBJECT TERMS

object-oriented programming
visual programming language

15. NUMBER OF PAGES

169

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL unlimited

SBIR Phase I Final Report

by
SNAP Technologies, Inc.
24 Upper Creek Road
Etna, NY 13062-0247

"Object-Oriented Visual Programming Language"
Douglas J. Davenport, PI

in fulfillment of
Contract Number DAAH01-95-C-R101
14Mar95 – 16Oct95

Issued by US Army Missile Command
AMSMI-RD-PC-GX
Redstone Arsenal, AL 35898-5280

Submitted 16 October, 1995

Project Objectives

An Overview
Research Focus
Specific Objectives of Phase I

"...There was also the fat that there were beginning to be more and more people who wanted to solve problems, but who were unwilling to learn octal code and manipulate bits. They wanted an easier way of getting answers out of the computer. So the primary purposes were not to develop a programming language, and we didn't give a hoot about commas and colons. We were after getting correct programs written faster, and getting answers for people faster. I am sorry that to some extent I feel the programming language community has somewhat lost track of those two purposes. We were trying to solve problems and get answers. And I think we should go back somewhat to that stage. ..."

—Grace Murray Hopper [Wexelblat81]

An Overview

During Phase I SNAP Technologies, Inc. was to undertake research to determine the criteria of a general purpose visual object oriented programming language and to determine whether such a creation existed in the commercial, academic, or development worlds. In its absence, SNAP Tech was to put forth a design for a general purpose visual object oriented programming language. As research progressed we recognized the additional need to research non-object-oriented visual languages, and did so. Several times we thought we had found *the* general purpose

visual object oriented programming language, but through extensive research and testing each time found that the delivery fell short of the promise.

As a result, we offer in this Final Report the SNAP Tech alpha design for both language and development environment, fulfilling and exceeding the requirements of the Phase I R&D effort.

In our Phase I proposal we recognized the value of expediting the [entire software development] process, making the solution methodology better fit the task, and finding a balance between what machine should do, what language should do, and what programmer should do. We refined these ideas to a single point: *productivity.* Provide tools to the programmer that include a robust core set of features and objects that provide extensibility, an environment that aids more experimentation, debugging, code creation, and maintenance cycles, make it dynamic and make it visual—and we have the basis for greater productivity.

During Phase I we researched answers and reasons for seven technical objectives, and pursued them via an eight step work plan. Summarized, we were to research and compile data on known visual programming languages (VPLs) including commercial, academic, in-progress, etc., then systematically categorize them, determine criteria for evaluation, analyze them based on these criteria, determine those components useful and usable in a visual environment, document these, describe the most important attributes of a general purpose (GP) VPL, then design an environment to support those attributes. Finally, we were to write the Phase I final report and if invited, a Phase II proposal.

The following pages describe in detail the specific objectives, the work undertaken, where, how, results, and our belief of the conclusion's technical feasibility for producing a general purpose visual object oriented programming language and environment (VOOPL/VOOPE).

No buzzwords!

In this report we have intentionally refrained from using jargon. Because the community has not yet determined clear, precise definition for many of these terms, we choose to explain ourselves with standard English phrases. We trust this lack of buzzwords will only clarify the results to readers of this information.

Research Focus

Our research sought a completely general purpose visual object-oriented language, existing in development or a more mature state in a commercial, academic, or available private enterprise as of this writing. The steps used for Phase I research were: 1) research and compile data for features, ideas, methodologies, etc., 2) record and classify according to the Burnett/Baker classification scheme [Burnett94], 3) create a basis for evaluation of this information, 4) conduct a language analysis, 5) determine specific elements of the environment appropriate for a visual programming language, 6) analyze appropriateness of features from existing languages, 7) determine the most important attributes for a general purpose visual object-oriented programming language, and 8) then design the environment for such a language.

This research included study of over 300 articles, books, papers and theses, more than 100 languages (finished and unfinished), and conversations with an assortment of designers and authors. Not all items used for research are included in the references. If the item's content was marginal to the topic, or did not meet our in-house standards of quality and reliability, the item does not appear.

An bibliography with abstracts and annotations is included at the end of this Final Report.

Specific Objectives of Phase I

Our Phase I technical objectives outlined a comprehensive, systematic approach to find, identify and classify, using the Burnett/Baker Classification Scheme [Burnett94], the existing and in-progress general purpose visual programming languages composing today's state of the art. In addition, we established criteria based on our professional experience and expertise plus the writings of other experts in the fields of object-oriented languages, visual programming languages, and interface design, against which to evaluate each language/product. Each criterion identified serve the same purpose: to create a programming environment and language that make the programmer and the resulting code more productive and more efficient.

By following these steps we examined these languages and considered the worth of each as a solution or contributor to the final design. In this paper we document our results, offering languages/products of any

significant interest shown in the Burnett/Baker Classification Scheme plus rating these languages/products against our own criteria.

The specific technical objectives from Phase I were:

Technical Objectives of Phase I

- Do any general purpose visual programming languages exist today?
- If not, what explanations exist for this lack?
- What execution paradigm is most suited for general purpose languages?
- What visual representation is best suited for a general purpose language?
- Can VPL be integrated with existing code written in a non-visual format?
- Why, historically, have visual programming languages addressed only specific tasks?
- What exists today that could be incorporated in some form for a general purpose visual language?

These objectives were met and exceeded through intense research. The answers to some of them—what visual representation is best suited for a general purpose language—, for example, are still in their infancy. Many experts agree, at least in this case, that only recently have computers become commonly available that are sufficiently robust to support graphics, and little proof yet exists that one representation is clearly better than another. Contention is strong among this same group, however, that the *implementation* of the interface is at least as important as is the actual *look* of the interface.

The lesson available here, based on years of related interface experience (even text-based experience), extrapolates cleanly to the broad scope of visual representation and beyond. The enduring, classic design guidelines are not dependent on each new hot idea; they are enduring because they work nearly regardless of platform, buzzword or trend. Some of these enduring guidelines are discussed as criteria for evaluation in the next section.

To arrive at definitive conclusions to others of the objectives, we formed a series of criteria based largely on the research findings and our language design experience. For the first objective—do any general purpose visual programming languages exist today—we obviously needed to consider the merits based on an impartial scale against a designer's conclusion. Thinking of LabView, for example, some feel it could be used for general purpose programming even with its very specific problem domain emphasis.

Another of the objectives—what execution paradigm is most suited for general purpose languages—requires an in-depth understanding and discussion, and that is provided in addition to our determination.

In all, the objectives from Phase I were thoroughly met. The detailed results are contained in the third section, Results Obtained.

Work Performed

Where and How
Research Conducted
Determining Criteria for the Language and the Environment
Concepts Extrapolated from the Criteria

Where and How

All research had a homebase of the corporate headquarters in Etna, New York, located in central New York. Through modem and network connections we probed the libraries of other states and other countries. Interesting research is taking place in New Zealand.

Logistically, we recognized four areas to target for research, each requiring a different method. For history, perspective, and recently published works, we sought published works. For academic papers we determined the pockets of greatest research interest and investigated the universities through telephone, on-line library searches, masters and doctoral theses, and personal communication. To find the latest, not-yet-published projects we investigated the moderated newsgroup comp.lang.visual. We determined who among the well-published people tended to co-publish and then sought out all authors' works. We also spoke with professors about our research. To find the commercially available products, both available and near-term, we used trade journals and comp.lang.visual's Frequently Asked Questions list (FAQ).

Professionally published works

Beginning with the professionally published works, we used keyword searches to find topics and authors from the world-wide Books In Print database. We also used every book recommended by the moderator of the most appropriate newsgroup, and every known published work of several key authors, Dr. Margaret M. Burnett primary among them,

plus. After obtaining and reading these books, we widened our approach by finding the works cited in *their* references. We did this until most citations referenced works we had found. We continued to monitor new publications through the end of the contract term, incorporating even books published in late 1995. This wide sweep may not have found every title categorically listed in the field, but it did find nearly every book of significance for our topics.

Academic works

The FAQ proved very useful for targeting where in the world pockets of research effort in the field were most productive. Using Internet we perused directories and information pages offered by these universities to see who were pursuing topics of interest to the project, then retrieved available working papers. We also sought information from the home universities of various professionally published authors.

In-progress works

Here the FAQ again was useful, as it is a forum for discussion. We monitored the newsgroup continually, noting updates and changes, and pursued leads with phone calls and other appropriate means.

Another successful method of finding new and developing work was personal networking, asking professors and other authors who was doing something unusual or particularly interesting. In general they were a helpful group, with Prof. Burnett even sending papers not otherwise easily obtained.

We contacted some developers directly to discuss their work. Using this approach we tracked even found designers who thought they were working in obscurity and who didn't necessarily want their work discussed. After explaining our reason for contacting them, no one wished to not be considered. One company required a Non Disclosure Agreement, which will, of course, be honored. As this particular company did not meet the criteria required to be considered "significant", no further information about them is needed.

Commercially available products

Demos, newsletters, downloads, white papers and manuals were all requested of commercially available products, with varying degrees of success. Some products of particular interest were purchased if no demo was available, with the knowledge that these were a "for approval" purchase. Of the commercially available items, those considered "significant"' are categorized along with the books and papers in the classification section.

Some commercial enterprises had no information to offer; phone calls to other results in a recording stating that the number was no longer in ser-

vice. One product recently acquired by a large company seemed caught in an administrative black hole on the product. Each phone call produced no product, but the assurance that "this" department had it. A demo was never found.

All books, papers, demos, etc. were reviewed and catalogued according to the Burnett/Baker classification system. This system was designed specifically for classifying papers, so we have use a subset appropriate to our work.

Research Conducted

The Research Process of Phase I

Our research sought a completely general purpose visual object-oriented language, existing in development or a more mature state in a commercial, academic, or available private enterprise as of this writing. The steps used for Phase I research were: 1) research and compile data for features, ideas, methodologies, etc., 2) record and classify according to the Burnett and Baker classification scheme [Burnett94], 3) create a basis for evaluation of this information, 4) conduct a language analysis, 5) determine specific elements of the environment appropriate for a visual programming language, 6) analyze appropriateness of features from existing languages, 7) determine the most important attributes for a general purpose visual object-oriented programming language, and 8) then design the environment for such a language.

The *expectations* of Phase I were that a visual object oriented language could result in: 1) reusable code, 2) an improved ability for programmers to comprehend large or complex bodies of code, and 3) shorter software development and maintenance cycles. The *criteria* resulting from Phase I all focus on designing a programming language and environment that makes the programmer more productive. Both object-oriented methodology and the visual componentare key to this goal.

First stated in our Phase I proposal and strengthened with every day's research is the belief that to create a general purpose visual programming language, one must begin with that as a goal. Each design innovation, every concept, every feature must be interwoven and synchronized with the others to provide a robust, flexible design.

Over the years, one method of extending a mature language was to add (or attempt to add) object-oriented methodology. With a few notable exceptions, what we found were examples of text-based languages given a graphical user interface and of procedural languages with

object-oriented methodology glued to one side In cases where object oriented features were added to mature languages, such as OO-COBOL, OO-Ada, OO-LISP, OO-Pascal, etc., whether procedural or functional languages, the result was to increase that language's complexity. .Managing that complexity then becomes the responsibility of the programmer, tasked now with learning complex idiosyncracies of a system trying to do something other than its original purpose. Since easy maintenance and creation are explicit goals, this is clearly unacceptable.

Recognizing that adding object oriented (OO) functionality to a mature language was a fundamental design weakness, we were especially interested in designs integrating all components from inception.

Some Pertinent History

Clearly, these add-on methods of building a new language do not offer good examples of software engineering or design. Yet this add-on method follows the normal process for software advances over the past fifty years.

An Evolutionary Process

In the early years of electronic computing, people actually manipulated bits—without the luxuries of a language or assembler. Grace Hopper, in her keynote address to the ACM SIGPLAN History of Programming Languages Conference in 1978, pointed out that early programmers made a quantum leap in thinking when they built a natural language front end to the machine level functions they had managed heretofore. People then overlaid instructions on groups of bits, and Assembly was born. This still required hand translation to bits, but it was progress. Eventually programs evolved to do all this, and the early languages were born. Time passed.

Then people wanted to work in equations...so came FORTRAN. Next, people wanted to work with less mathematical objects, such as names and addresses, and then came COBOL. Abstraction progresses, and people begin adding libraries of complex mathematical equations. More time passes. Along comes object oriented technology, a farther abstraction still, with even more abstract data.

Most, if not all, of the languages in use today are derived from FORTRAN, COBOL, ALGOL and LISP. And the vast majority of these languages in use today were developed to "fix" some part of one of those languages. Software development is the slowest aspect of technology. Hardware changes drastically in short periods of time; radical design changes result in radical levels of performance improvements. C, the programmer's staple, has been around for 25 years! Changes are coming

more quickly in the last five years, and interesting moves are being made in visual languages and in object oriented languages.

Understanding this evolutionary process sheds light on the lack of a general purpose VOOPL. With very few exceptions, the existing environment/language paradigms are not appropriate bases for its development. This very different paradigm requires a leap of logic to a new frontier of thinking and creation. Reasonably good languages exist for specific purpose VPLs, but extrapolating from them to a general purpose language is not a practical nor workable approach, although tried frequently.

These extrapolation or add-on approaches *will not* accomplish an order of magnitude improvement, and that's our goal. We propose to use everything learned in the last fifty years—we intend to combine features from several languages, chosen for their appropriateness in meeting the productivity goals, re-cast them in light of a visual paradigm, then provide the features with a live environment to make the best use of each advantage. This clean, tight design will result in application code that will run on yesterday's microcontroller or tomorrow's workstation.

Determining Criteria for the Language and the Environment

The Criteria

The criteria were determined through research, through gaining insight into existing language/environment paradigms and their inherent strengths and weaknesses. The following paragraphs describe our primary and secondary levels of criteria. Each is labelled. The criteria eventually used to evaluate the research were:

Primary

- Suitability for General Purpose
- Consistency
- Views
- The Look of an Object is Part of that Object's Definition
- Simple, Expressive Language

Secondary

- Object Support Design Designed in/Built into Language
- Allow for Special Cases within General Purpose

Suitability for General Purpose. (primary criteria)

When a language is specifically designed for certain functions, it tends to do those functions well. When that specific-purpose language design is exploded to general purpose functions, that may not enhance the major design strengths of the language. Here's a case in point. From the words of an abstract, "Although inteded for use in symbolic computation, this language should prove interesting as a general purpose language." [Baumgartner90] Yet these strengths in one discipline may offer strengths for another. When considering any specific purpose language, we considered how well it could be used for other diverse purposes...for word processing, for simulation, for example.

Consistency. (primary criteria)

Whether primitive or composite, objects should interact with the environment in the same manner. This simple statement is actually incredibly hard to enact, yet it remains a primary criterion. The goal, again, is to produce a language which offers consistent use from bottom to top, inside and out, with an end goal of greater productivity. *Relearning* action/reaction, interaction, etc., directly negates that end goal.

Views. (primary criteria)

First, a word about people. People recognize and categorize objects and concepts based on their own lives, their own personal paradigms. Second, a word about views. A view is essentially a glimpse into the language. Now, recognizing that people understand things differently as a result of looking at the same picture, we sought ways to present information so all who saw it would reach the same understanding through their own paradigms.

We reviewed products that offered one view, a "take it or leave it" approach. Others offer different angles of the same view. Some offer a view of the operating system, some show the human interface, and some show code or data structures. Since people comprehend information differently, we believe that flexibility in views is essential.

We want to be able to restrict and expand views, for example closing the visual of a loop and seeing it represented by a different, "closed" visual. We have not determined the optimal number of visuals to make up any one view; recognizing that fewer is at least simpler, determining the minimal number workable for any given situation or application seems a good start. That determination, optimally, would be made by each individual user.

Overall, we want enough different views to understand changing perspectives and to focus efforts without visual noise. For example, some people need to see data in a flow chart, others in a data flow diagram, others in hierarchical form.

This concept of view is a critical element for both code creation and later for code maintenance and the development of legacy systems. We believe that views must be flexible enough to allow a programmer to see what they need to see to understand the code.

Further, we determine that a programmer should have control over how they want their view to appear. This simple-sounding statement is a burst of freedom compared to anything we've found or experienced.

Finally, views of a dynamic environment are, by definition, animated. Dozens of papers argue that animation is the logical growth for a visual language, and we cannot imagine trying to represent dynamic interaction without animation.

The Look of an Object As Part of that Object's Definition (primary criteria)

In most object oriented languages objects are defined by data and the action which it performs on itself (or on the data it contains). In a visual language, an object can be identified in at least two ways: all objects could look alike and each must be opened to understand its contents, or the object may be represented graphically by its behavior. Less mental effort is required to recognize the graphic. When the object designer also defines a look for an object, that makes using the object easier for maintenance and reuse, and also for debugging.

We are interested in not only the look of an object, but also in letting the user redefine that look. This reflects the need to let objects be viewed in a method most meaningful to the use.

"A language that doesn't have everything is actually easier to program in than some that do."
- Dennis M. Ritchie

Simple, Expressive Language (primary criteria)

The language with fewer features can at the same time be a very expressive language because it is easier to learn and to use. A simple structure, unencumbered, relatively free of idiosyncracies, lends itself to more rapid understanding, cleaner design, and more full user.

Object Support Design Designed in/Built in to Language (secondary criteria)

Beginning with observations from experience, we note that most programmers with whom we interact tend not to use most or even half of the features of an object oriented language because they do not believe that managing the complexities reaps a value worth the effort.

We believe that object oriented designs integrated from conception will be simpler, more elegant languages allowing programmers to be more fluent, and to use the language to their fullest extent. Adding features and add-ons creates levels of complexity and idiosyncracies which create a need for more learning, resulting in greater difficulty for the programmer. The superior design results when a software designer need not maintain non-object oriented objects or features of the base language.

Allow for Special Cases within General Purpose. (secondary criteria)

We recognize that a truly robust general purpose design will include the flexibility to accomodate special case needs. The primary special cases known that have significant and unusual requirements are realtime and embedded systems needs (garbage collection features, lengths of time needed, ability to determine lengths of time needed, accessing add-on hardware, etc.) As these are of interest to both commercial and government sectors, they are recognized and considered.

Also considered...

Less formal than a criterion but a factor watched for and considered was whether or not a language/application required system resources not commonly available. This is occasionally the case among the most interesting languages, and was not considered a detriment; it was simply noted. Hardware improvements come quickly.

Concepts Extrapolated from the Criteria

A Dynamic, Live Environment

A visual language leaves behind many of the encumbrances of a text-based language. In the text-based paradigm language was one world and environment was another world. The programmer worked in a language, character by character, creating code. The language world was then closed, the environment world opened, the new code compiled, errors reported and analyzed, and then the environment world was closed in preparation for returning to the language world. The worlds abutted; they did not overlap and they did not integrate. Enter new world, Visual Paradigm.

Animation = visible change to graphic "code"

A paradigm shift is required to fully understand the vast ramifications of the inter-functionality of the language and environment in a visual setting. The language is now a *door* into an environment—an environment that one never leaves. The environment provides a picture of the computational state of the world—a changing, dynamic, live view of the world. As the programmer works, moving an object into a container object, for example, the container object visually and actually changes to reflect the change to "code". On-screen, without leaving Language and entering Environment, the programmer sees change and effect.

Incremental Construction

This live environment provides the programmer a means of "trying it under construction"; the environment promotes experimentation of technique, of learning and improving. This *incremental construction* is a key concept; one doesn't leave the environment to execute a function—the programmer sees the change dynamically and reacts accordingly.

This live environment is one of the rudiments of this visual paradigm. It has significant value as an experimental tool. The programmer has a ready aid for questioning, redesigning, rethinking, experimenting, prototyping. In a text-based world, such endeavors required major construction, major task control, even destruction before "renovation" could begin.

Think of a large, complex set of code as being analogous to a house. The code needs a widget updated; the living room needs new paint. In a text-based world, start by destroying the house. Rebuild with new paint. In a visual paradigm, a dynamic world, simply change the wall color—no destruction is necessary to experiment with change and to view the results. Experimentation (and learning) is fluid, implementation is accomplished by the experimentation, and it's done before the programmer's eyes, literally.

How is this accomplished with any of the known execution paradigms? This dynamic environment requires features of both procedural and functional execution paradigms. It requires parts of both, yet mimics neither. As is true in the features list, the execution paradigm begins with complementary existing features from disparate sources. Creating a revised execution paradigm requires prototyping, testing, evolution. The first iteration will include features to support the known criteria and features determined to be critical to this visual world. It is very likely that the mature prototype will entail a far greater number of support mechanisms than is now deemed necessary.

To summarize, in a live environment the program is always running. Traditionally, source code is generated, the program is run, the source

code is "revived" for revisions, goes dormant again, etc.; this, in contrast, is a dynamic environment. This never goes dormant, never stops being an interactive environment. The target is again simplicity and improved productivity.

Strength = Inherent Weakness of a Limited Problem Domain

Many of the most successful visual programming languages of today, such as LabView, share the limitation that for all their power, they operate in a limited problem domain. LabView, as an example, allows the programming of data acquisition equipment.

Besides being visual languages, these programs are based on the reuse of software modules. They provide the user with a library of components with well defined inputs and outputs. The user constructs a program by instantiating objects from the library and binding the objects together by connecting inputs to outputs.

In addition, one finds similar applications that inhabit the borderline between programming languages and games. An early example is Bill Budge's Pinball Construction Set; a more recent example is SimCity by Maxis. In both cases the user constructs an environment visually, and then activates a dynamic mode in which to play.

Finally, one may consider programs such as Adobe Systems Incorporated's Illustrator which is a visual front end to the PostScript programming language. Rather than writing textual code, the user draws screen images, which are automatically translated by the program into the underlying program text. This text file is later transmitted to a printer, where an internal computer executes the program to generate the image.

It is no coincidence that the limitation of these programs is also the foundation for their strengths. Because they operate in explicit domains, they can define a reasonably complete set of high level tools or modules.

Reuse

Visual programming languages are particularly suitable for metaphors based on apparent direct manipulation of tools. The mouse acts as an extension of the hand, and the screen presents the appearance of a workbench. Because our ability to recognize patterns is linked in some way to our visual intelligence, a visual programming language offers the potential for high level representation of program structure. These factors imply an approach based on representation and manipulation of high level modules. Once the programmer has become accustomed to manipulating modules, work moves to a higher level of abstraction. And once this begins to happen, reusable code becomes a reasonable possibility.

Research suggests and experience confirms that simply providing the *opportunity* for reuse is not enough. Modules and libraries succeed in creating high level abstractions through information hiding. In spite of this capability, reuse is less common than one might expect. We believe that the failure of reuse is due to the absence of any standards for constructing the module interfaces. Because programmers and module providers share no common standards, the programmer may find that the interface is too awkward or difficult, and in addition typically finds that the module does not match his needs well enough, so concludes that the procedure should be built from scratch.

It is instructive to consider a few domains in which libraries are widely used. The ANSI standard C library provides widely used functions for interfacing with file systems. In this domain, programmers share standard abstractions of suitable operations: one needs to determine if a file exists, to open a file, to read from the file, and to close a file, just to name a few. The library provides procedures to support those operations. Although the programmer must learn the exact specification of a procedure such as fopen, the abstraction that the procedure implements is already known. That is part of the training.

Another example comes from mathematics. Whether we consider common functions such as sin, or more complex ones that might compute the probability associated with some score or carry out some non-linear curve fitting, we find functions that have well defined and commonly shared abstractions. These functions are commonly reused not because of information hiding or because of evidence of particular brilliance in the specification of their formal parameter lists; they are reused because they implement common, shared abstractions. These abstractions are rarely, if ever, formally standardized.

It seems reasonable to conclude, then, that reuse is neglected not because of any failure in the capabilities of current programming environments; nor would consideration of this evidence lead one to expect that a visual programming language would through some technical magic increase the degree of software reuse, any more than object oriented programming has accomplished this. Reuse will occur as a result of sharing high level programming abstractions.

Results Obtained

Findings: Answers to Technical Objectives, Phase I
Findings: Items of Significance
Results: The Burnett/Baker Classification
Results: The Evaluation
Design Features for a Visual World

Findings: Technical Objectives, Phase I

Do any general purpose VOOPLs exist today?

Our Phase I research found a few general purpose visual object oriented language/products. None met all criteria established for the project. Of these, one is commercially available and another is an academic project. The commercial product is Prograph, available through Pictorius, Inc. and widely considered "the" commercial GP VOOPL. Prograph covers many of the bases for general purpose needs, but its implementation does not seem to have had productivity as a goal. It is included in the list of significant findings and discussed there.

The academically developed language is Forms/3 by Margaret Burnett, now at Oregon State University. [Burnett91] This is a language of many significant developments; in light of this project and its criteria, Forms/3 focuses more on instruction in computer science than it does on becoming a commercially viable product. It certainly provides significant contributions to the state of the art.

Many other languages and products and development projects were considered; these are the only two that were visual, general purpose, and object oriented. SELF is a very significant language originally from Stanford University [Ungar91]; it met many of the criteria but was text-based. SELF is worth watching in the coming years.

The Phase I Final Report includes a section classifying the projects reviewed according to the Burnett/Baker classification system. It also includes more descriptions than appear here about where other projects fell short of the goals.

Why not more GP VOOPLs?

The answer includes many good reasons. We believe the single most important reason that no truly general purpose VOOPL exists is that few commercial entities would undertake its development in a way that would ensure success, success being defined as a robust language that could survive the tests of users and time. Further, few commercial entities could afford the R&D of a general purpose VOOPL because the world isn't demanding its existence. Eventually the world may recognize the value of a truly general purpose language, with which they can build any tool required and have the solid support of a commercial enterprise behind the product—but Pictorius is living proof that a better mousetrap doesn't guarantee success. Pictorius is the maker of Prograph, a reasonably general purpose language and one of the items listed "of significance". The company has suffered severe financial difficulties, folded, re-emerged, and continues on. It has a loyal following, but the costs of staying in business are a continuing strain.

We do not expect to see a general purpose VOOPL emerge from academia simply because most languages in that setting are developed as a research tool or interest. Burnett leads the group interested in developing a general purpose VOOPL, and speaks clearly of the problems still to be overcome. Her interest in the general purpose version is still in the minority, despite Burnett's long list of publications and her professional stature.

Also, until the last five years or so hardware was not really sufficient to perform the realtime calculations and display required by this environment. Working in a visual environment that requires minutes to display an update isn't conducive to software development.

And as mentioned earlier, the nature of software development leads in the direction of tweaks and updates and fixes. A *fundamental* paradigm leap is required to conceptualize and begin to design a general purpose visual object oriented environment.

Execution paradigms

An interesting question, and again a complex answer is required. Without making any assumptions, each paradigm was considered from many perspectives. Each was viewed in light of what it could offer, what strengths and weaknesses it would present, and the overall suitability. In short, nothing fit perfectly. The paradigm proposed in the following

pages is a mix of known paradigms, with the exact proportions from functional and procedural still under consideration.

Visual representation

A comment attributed to Michelangelo comes to mind here. When asked how he brought forth such spectacular sculpture, he replied that he merely saw the shape within the stone and carved away everything around it. Carving away at the edges of the unnecessary clutter is indeed an essential task in visual representation.

We know that even the best of graphic images are perceived differently by different people. Society, culture, age, gender, occupation all influence the paradigm through which each of us views and defines the world. Recognizing this unavoidable variation, we present multiple ways of viewing this language/environment. Custom views, dynamic views, and more are discussed in detail in following chapters.

We know that color is useful in many graphics. We also recognize that the misuse of color, as well as pattern and shape and size can be confusing and troublesome. Each component of the visual interface will be introduced only as it aids in recognition, aids in consistency, aids in production.

The overall outcome of this Phase II project is a VOOPL whose gains in productivity over other existing language/environments should be dramatic and significant. The in-house goal for this improvement is an order of magnitude. This requires creative, innovative use of the most appropriate features and superb core design. That leaves little room for pasting on an interface that falls short of implementing similar leaps of design.

Integrating with non-visual code?

From the beginning of designing the SNAP Tech version of this system, a unanimous decision among the design team was the agreement that older generation code must be recognized, addressed and its inclusion/incorporation accounted for. We do not suggest that non-visual code brought unchanged into this new paradigm will perform at the same level of speed and efficiency as code created in the system, but we promise that at least at some levels existing non-visual code can be brought into this model.

Why only specific tasks vpls?

Two reasons exist for this situation. One references the historical evolution of software development—with a few changes, an existing language could now address a specific additional need. Hence a specific task VOOPL. Those languages designed specifically as a vertical market

fit, such as LabView, draw strength from their designs which are all attuned to performing market-specific tasks extremely well.

Another reason for the existence of specific task VOOPLs, gleaned from research, is that these have been developed by academicians to explore or teach specific computer science situations. As these are developed with a very particular focus, they tend not to be good bases from which to extrapolate to a general purpose paradigm.

Available now to use in a GP VPL?

Good ideas abound. The critical skill in researching the ideas is to understand what will work in another paradigm, and what strengthens another concept. Some languages use a fairly large (300+) set of core objects; others use as few as a dozen. Both ideas have inherent strengths. Some languages use a dynamic environment but without a visual interface. Can a visual dynamic environment be made to work at all? On today's computers? On yesterday's computers? On any CPU conceived at all? Some languages offer features that are fluent, elegant in their own paradigm that are not at all transportable to another paradigm.

The innovation, the genius, is in extrapolating to create a new and clean design, robust, integrated, spare. The genius is in building a new unit, composed of new and existing units, with the new an order of magnitude stronger together than any unit separately.

Findings: Items of Significance

Items in this list may be a language, a product, or an environment. The list is in descending order of importance, with the most significant being first.

By inclusion in this list, the item has met a substantial part of some or all criteria established to be deemed "significant". In some cases the overall item is of little interest but some concept within is a contribution of such importance that the exclusion would be unreasonable.

SELF. This is the single most significant and interesting language (or product) found in the course of Phase I research.

text based with a visual front end. fully object oriented. uses dynamic inheritance.

This is a language, like Smalltalk in this sense, specifically designed *not* to write programs, but to study the *process* of writing programs. It is a fully object-oriented language (meaning that the object-oriented features were designed in from the beginning), and uses the prototype object model instead of the class-inheritance model found in Smalltalk and others.

Self is a simple, reasonably consistent language. The only other language comparable for simplicity is Lisp, but Lisp is not a pure object-oriented language.

Self appears to have originated at Stanford University and now has alliance with (or is the property of) Sun Microsystems Ltd. and Stanford University. Look for references [Ungar91] [Agesen95] [Wolczko95] [Ungar95] [Self95] [Madsen95].

Drawback:

- the reduced performance resulting from features such as dynamic binding, automatic memory management, etc.
- text-based, reducing the expressiveness of code.

Noted, but not particularly a drawback:

- requires extreme system resources; Self requires a very major workstation to run.

Significance:

- simplicity
- visual interface is fairly clean and consistent.
- This language comes the closest to meeting all criteria of any found through Phase I research.

Vista. visual, object-oriented.

Implemented in Smalltalk. Vista visually manages to incorporate and integrate data flow and control flow (transformational systems) successfully, superbly. This is a great advantage for a language because it better fits a much wider variety of problems.

One flaw in the visual environment from a productivity viewpoint is that there is currently no visual environment for creating objects—the programmer must leave Vista and return to Smalltalk for creation. This

imposes all the restrictions and constraints of using a text-based environment.

Drawback:

- poor performance
- no visual environment for creating objects
- dependent on Smalltalk base

Significance:

- elegant visual interface
- superb *incorporation* of two paradigms

Forms/3. object-oriented, visual, based on forms. Developed by Margaret M. Burnett, currently at Oregon State University.

Forms/3 is a declarative language where objects are created by designing a form. The author chooses language features specifically designed to avoid some fundamental computer science concepts, which leads to the belief that the emphasis is on making the inexperienced user more productive.

Drawback:

- the environment is not ideal for production use
- while not a focus the development effort, Forms/3 requires extensive system resources
- while in a sense this is a general purpose language, it is more general purpose in an academic setting than in a production setting

Significance:

- dynamic environment
- objectness is prototype-based

Prograph. only commercial general purpose visual object oriented programming language on the market.

Prograph is a dataflow language, as is LabView, and both are commercial languages. There the similarity ends. The interface is a significant problem with Prograph, although the problem may have been deeper and also reflected implementation problems with the language. The interface was so bad that it was difficult to probe deeper. At the same

time, offering a good interface for anything more general than a limited problem domain is a challenge in the field.

Drawback:

- very difficult to use, even with a significant investment of time
- currently single platform implementation

Significance:

- commercial

LabView. visual, object-oriented, limited domain environment, commercial

LabView offers tools for virtual instrumentation, data acquisition, and minimal data manipulation. While it has some general purpose features, they are not its strength and building a large or complex program would not be particularly pleasant.

It is a dataflow language with sufficient speed for the tasks of the environment. The developers have focused heavily on improving the speed of the language, and continue in this effort. Clearly, that is something of importance to the National Instruments, owner of LabView.

Its visual interface works smoothly, easily. Unlike Prograph, LabView's interface was an aid and was well implemented.

Drawback:

- very limited problem domain

Significance:

- extremely good in its problem domain
- commercial
- easy to use, even by people with only vague understanding of the problem domain

C++. non-visual, general purpose.

Currently very popular, C++ is probably at this time the most heavily-used (and *under*-used) language in the world. Developed by Bjarne Stroustrup, C++ is an extension of C.

This popularity offers interesting possibilities: one is an inclination to develop a visual interface for C++ simply because of the large programming community attracted to the language. The inherent complexity of the language, discussed below, offers an unsuitable foundation for this. Remember that this language is already an add-on version of the foundation language.

The other would be to create a visual language using C++'s execution paradigm. This is analogous, for example, to labelling musical notes A, B, C, etc., not because those notes represent an A, a B, or a C, but because many of us recognize those entities and can more readily then understand something with that same interface. The result of creating a visual language using C++'s execution paradigm would not be a better language; it would be a language with the complexities of C++ with additional idiosyncracies required to offer in a visual paradigm every function of C++. This is not a pleasant prospect.

As stated, one of C++'s most notable features is it s complexity, which is due directly to its evolutionary history. As a result of this complexity, C++ is also a greatly under-used language. In an attempt to maintain compatibility with C or to prevent adding additional keywords to the language, C++ has idiosyncracies that create significant complexity for the programmer.

Drawback:

- not visual
- complexity

Significance:

- overwhelming popularity.

Lingua Graphica. visual, object oriented, 3D

This is a visual language built on C++. The language was designed for virtual reality users who needed to work on code while in the VR world. This language illustrates the case mentioned in C++; the visual front end

is unlikely to make C++ users more productive, but in this specific paradigm the language is a good fit and meets specific needs.

Drawback:

- inherent performance problems of adding a visual interface to C++
- inherent increased complexity for the user

Significance:

- a good fit for the need.

Ada. text based, in a manner similar to C++.

Although never sharing C++'s popularity, Ada is in some aspects a technically superior language. Like C++, it has problems with complexity issues although from different sources.

Adding a visual interface to Ada would result in many of the same problems faces in adding a visual interface to C++. We believe that programmer's productivity would not be significantly improved, perhaps as little as a 20–50% improvement. We also predict a high learning curve, again partially based on the complexities.

Drawback:

- not visual
- complexity

Significance:

- technically superior to C++.

Smalltalk. text-based, pure object-oriented language

Developed by the Xerox Palo Alto Research Center, Smalltalk has the distinction of being one of the few languages with object-oriented features designed in from the beginning. This purity results in a language more greater consistency in use than others. The language also employs features such as dynamic binding that result in a somewhat poor performance. This makes the language unsuitable for many real-world uses. The performance problems are, in fact, significant enough to contribute to Smalltalk not being commercially viable.

Drawback:

- poor performance
- text based

Significance:

- pure object-oriented
- referenced commonly by academicians.

Lisp. text based, dynamic environment available, symbolic computation, artificial intelligence.

Lisp has been around for decades; it is the foundation language for literally dozens of other languages. It has very simple syntactic requirements. In its own evolution it has seen the addition of object-oriented features, while maintaining the consistency. It has evolved into a very rich, powerful language. Like Smalltalk it suffers from the performance degradations caused by features such as dynamic binding and automatic memory management.

While this is inaccurate, Lisp has long been considered a tool primarily for artificial intelligence. In fact it is reasonably suited to general purpose work; the public relations has much to overcome.

Drawback:

- never seen commercial success, partly because of its perception as an artificial intelligence language

Significance:

- overall contribution to computer science
- dynamic environment available
- simplicity and consistency of interface

—end of list—

Results: The Evaluation

Among the items of significance discussed none exhibits the degree of integrity to the full criteria that a truly general purpose VOOPL can do and will do. The value offered by each of these offerings is not in each as an entity; it is in the new execution of concepts when combined with other significant concepts to make a new whole.

The criteria, again, for a general purpose visual object-oriented programming language as determined by SNAP Tech are these:

- Suitability for General Purpose
- Consistency
- Views
- The Look of an Object is Part of that Object's Definition
- Simple, Expressive Language
- Object Support Design Designed in/Built into Language
- Allow for Special Cases within General Purpose

The key concepts understood early in the project, refined throughout the project, and echoed in the research, are the result of the entire history of programming language development. The embodiment of the ideal doesn't yet exist, in part because only so recently has the development community come to have an understanding of what the ideal may be.

The concepts underlying the development of the SNAP Tech model are these:

1. Focus on the ultimate goals and never lose integrity with them.
2. Provide a complete language, providing solidly for creation and maintenance.
3. Provide and demand consistency in every phase of design, in every form of implementation.
4. Lay the groundwork for elegant simplicity. Avoid anything extraneous.

The criteria used as a basis for the SNAP Tech design built on those first criteria and added a critical element: every design component had to support and strengthen every other component. Not sufficient was the idea that components could co-exist; they had to meld, producing a stronger result than any could offer individually.

Design Features for a Visual World

The objects in this world, beyond the primitives and core objects, have potential to become extremely complex. The add-on objects and eventual applications resulting from them will be an order of magnitude more complex...and these must be represented visually...in a dynamic environment. An interesting task. To accomplish this, the software design team provides these solutions: multiple, dynamic, and custom views of objects; ultimately consistent use and interaction at every level of use; and annotation capabilities appropriate to a visual paradigm, each directed specifically toward providing the programmer with tools for efficient creation and maintenance of code in a visual paradigm.

A feature frequently discussed in the literature and used by other developers as a criterion is specifically *not* part of our design: the intuitive look. The SNAP Tech design team has determined that no one look is intuitive for a broad scope of people, and that this fact negates any practical implementation of the idea. Instead the team focuses on providing enough views, enough customizable, flexible views, for each programmer to create the view that shows what the programmer wants to see. Just as a programmer will see the cause and effect of actions (changes of state and behavior), the programmer can determine how those changes will be displayed visually.

The criteria determined to be influencing factors of the design are: dynamic views, multiple views, custom views, consistent use/interaction, and annotations.

Dynamic Views. In this live environment, changes are visible as they occur through the medium of a dynamic view. In this visual paradigm, this makes perfect sense. Imagine pouring milk into a glass. The visual result influences the next behavior...when the glass is full one stops pouring milk. This simplistic analogy fits well in a debugging mode: when an action is interrupted because of a particular change in behavior, the programmer sees the interruption as it occurs and is likely to see the cause as well as the effect. The immediacy of the live environment—an action causes a change, and the change is visible—is brought to full use when the programmer watches the code perform.

In more precise language, a visual representation of an object shows that object's current state. For example, an empty generic container object displays as an empty rectangle. When an object is added to the container object, the empty rectangle may appear with a small circle inside it. As the programmer drags the circle into the container, this is displayed, live, real-time.

Programmers working in a text-based system have the option to write a line or two of code, then compile and debug. This is seldom the working methodology. Programmers usually write a significant amount of code, likely a fairly complex set of code, before compiling and debugging. The likelihood of complex error is much greater this way, but seems to be a preferred method for many programmers. This visual example works well in a simple situation, and its value is even more profound in a complex application.

Multiple views. This is the SNAP Tech answer to an intuitive look. What's intuitive to one programmer is likely not to be indicative to another. After reviewing papers describing at great length the intuitive nature of the interface, our team is in agreement that There's No Such Thing as an intuitive look for something as complex as programming code. Multiple views, however, addresses the same issue and works.

Filters are a function of multiple views. Filters are a set of collection criteria that selectively show or hide a given set of objects. This allows the programmer the ability to focus into a specific place or into specific actions. A filter applied to the same code through any view should yield the same results.

An example will help. Programmer A understands problems and concepts in light of Paradigm 1. This person interprets existing code and new needs within that framework—that's how they think and what makes the most sense. Programmer B has a different paradigm, and recognizes solutions best in terms of the personal paradigm. Either could adapt over time, but both are most effective when allowed to work within the paradigm of their choice. Multiple views of the same code allow this flexibility. Multiple views allows each programmer to view the state and process from their own base of experience, making each the most immediately productive.

Custom views. Part of an object's identity is the definition of its view, the graphical physical representation. In addition to a generic view, each object may have multiple custom views.

This is best explained through an example. To begin, a rectangular container object is empty. A common view of this object is likely to be an empty rectangle, color and size as yet unspecified. Into this container object are added one triangle and one very complex object, circular overall. The programmer will use this filled container object many times in construction of an application, wants the new complex object represented simply, so chooses a particular view that expresses the new functionality. This view is assigned as the "default" view, actually that

programmer's personal custom view, for this new object unless a different default is selected.

Orthogonal design, Consistent use/interaction. Consistency is a rare thing, perhaps because it is so difficult to achieve. Software designers operating in a visual paradigm recognize the need for consistency, for simple, stable rules. Bruce Tognazzini [Tognazzini92] writes, "The same class of object should generate the same type of feedback and resulting behavior, no matter in what part of the program ... they appear". In other words, using the house analogy again, an "open" function must be consistent in its use. Whether opening a door, opening the refrigerator, or opening a window, the operative word is *open*.

Consistent action requires less memorization, which in turn aids productivity. To make an action easier to remember, simplify it.

Orthogonal design refers to the clean, linear relationship of all parts. The whole could not function effectively if any piece were missing. This level of design elegance never occurs by accident, and is nearly as rare as consistency. It is an appropriate goal for a language and environment expected to be the source of a new development paradigm.

Annotations. In a visual language, text-only comments are not sufficient for clear or complete comments. The software will support graphics, fonts, etc., thus providing a means for meaningful, accurate comments and literate programming.

Estimate of Technical Feasibility

The Outlook
Why This is Attainable
***How* This is Attainable**

The Outlook

Producing a general purpose visual object-oriented programming language is 100% achievable at this point in time.

We project, however, that the development of this VOOPL in a timely manner will only come about when commenced as a commercial effort, based on solid software design engineering principles and mindful of past effort, accomplishments, and pitfalls.

Why This is Attainable

As a concept grows from infancy through maturity it encounters many difficulties. At first it's awkward, doesn't know how to express itself, and may not be well understood or well received. Its environment may not be suitable.

As it moves to young adulthood more of us recognize its value, even in an unpolished state. We think about it, we brainstorm, and we tinker. We share our research, we finetune the ideas, and the concept begins to be more readily understood, its value more obvious. The environment matures around it.

The documented roots of visual programming languages is thirty-some years old. Thirty-some years ago the hardware was insufficient to the task. That's changed.

Over these thirty-some years the topic has reached a certain stage of maturity—the development community understands the concept, and significant research has investigated many issues. We understand more, and we now understand more of what must be overcome. Some of us feel that no single existing paradigm is sufficient to this new task; just as the language and environment will evolve together, so will the paradigm. As will the eventual mature language, the paradigm will incorporate some of the best of the existing knowledge with some particularly creative new thoughts.

The development community seems to be moving toward agreement on a broad conceptual base of goals. That's progress. We have a plethora of specific purpose visual programming languages from which to learn. We have a wide variety of very solid programming langauges with dynamic environments, object-oriented technology, etc., from which to learn. And we have opportunity.

So we come to a point in time when skilled developers with a wealth of language experience, language design skills and interest, and a recognized need meet. That's progress. And that's attainable product.

It is our strong belief that the timely development of a general purpose VOOPL will come from a commercial enterprise, familiar with the rigors of budgets, milestones, deadlines, R&D, deliverables, and the long-term commitment of bringing a product to market and its lifetime support.

HOW This is Attainable

Several components must be working in near-perfect collaboration for this project to come to fruition technically. Designers must comprehend at an intuitive level the goals of the project. They must understand the areas and scope of technical difficulties involved. The depth and scope of experience and expertise with software engineering principles must be exceptional. Similarly, they must be adept at finding or creating solutions to those difficulties. For this, a broad and solid understanding of existing research is invaluable. And at a basic level, the design team should have a very wide base of programming language experience and knowledge.

We believe the development should not occur in a vacuum. While intellectual property rights must always be protected in a commercial venture, as soon as possible an early view of the product should be available to interested parties. The comments and suggestions are useful to designers. While not deterring from the original goals, these ideas should serve reinforce the *general* perspective and provoke additional brainstorming.

The skills to develop this language are likely to exist somewhere in academia right now...amid teaching responbilities, publication requirements and departmental meetings. The urgency does not exist, overall, in the academic world to produce a finished general purpose VOOPL. The commercial perspective is critical to this project's timely development. From the commercial sector comes an emphasis on producing a useable, quantifiable, justifiable product through significant attention to original design, revision needs, prototyping, milestones, deadlines, and budgets. While this is in a state of change, the academic environment overall has investigated points of object-oriented technology, visual programming languages, animated icons, human interface issues, etc., as research topics nearly sufficient unto themselves. A wealth of valuable research is accumulated this way; bringing this research into the commercial sector, assimilating the information, recognizing its value in the original and in new settings, and producing a distributable product—these are the strengths of the commercial sector.

The R&D must include continually monitoring world-wide progress in appropriate fields. Established personal conversation and information exchange established with the research community should continue.

Then with the technical issues recognized, a solid commercial support structure must be established to support the continuing efforts. Briefly, to supplement the technical issues, the developer should:

- determine a workplan culminating in mature, working prototype with sample application
- establish milestones, deadlines, timetables
- establish structure to develop interest in project (via WWW, newsgroups, etc.)
- support the business structure throughout the development phase of the project, including investing in staff, equipment, etc.
- establish structure to support launched product (email, WWW, telephone) w/staff, info db, etc.

- construct a business plan describing company subsidy available through development, plans for distribution mechanisms, income generation, follow-on product development, growth, sustainable reinvestment and continued development

Further, from early stages onward, designers must consider the importance of eventual acceptance of their language by national and international standards committees. Languages are written, Grace Hopper would remind us, to allow people to converse with computers. If a language proves its worth over time, it is considered for acceptance as a standard. It is our opinion that for a language to be considered a long-term contender, an essential requirement for large and complex systems, such acceptance is very important. Recognizing the elements of endurance and incorporating them into the original design goals is a worthwhile task.

In summary, the project is completely technically achievable. It is a task of some difficulty, but the essential elements to produce a mature, working prototype and eventual complete solution are in hand. The design goals are well understood, the first-round design is well underway, problem areas are understood, today's hardware is adequate (although bigger and faster is always preferred), and creative, skilled minds are available and sufficient to the task.

Languages and Products Reviewed,

Classified using the Burnett/Baker [Burnett94] Classification Scheme

Ada — Object-oriented, Text based, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
  - F. Exception handling
- VPL-V. Language Purpose
  - A. General-purpose languages

Anim3D — Visual, Text based.
- VPL-I. Environments and tools for VPLs
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - B. Control flow
- VPL-V. Language Purpose
  - D. Scientific visualization languages

ARK — Visual.
- VPL-I. Environments and tools for VPLs
- VPL-V. Language Purpose
  - A. General-purpose languages

Artkit .
- VPL-I. Environments and tools for VPLs
- VPL-V. Language Purpose
  - E. User-interface generation languages

Authorware — Commercial.

AVS — Commercial.

C++ — Object-oriented, Text based, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
  - F. Exception handling
- VPL-V. Language Purpose
  - A. General-purpose languages

C^2 — Visual, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - B. Control flow
  - C. Data types and structures
- VPL-V. Language Purpose
  - A. General-purpose languages

CAEL — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - B. Control flow
  - C. Data types and structures

Cantata — Visual, Text based, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
  - B. Visual representations
    - 2. Iconic languages
- VPL-V. Language Purpose
  - C. Image-processing languages

Capsule — Commercial.

ChemTrains — Visual.
VPL-II. Language Classifications
A. Paradigms
11. Rule-based languages
B. Visual representations
1. Diagrammatic languages

Cube — Visual, Object-oriented.
VPL-II. Language Classifications
A. Paradigms
5. Functional languages
B. Visual representations
1. Diagrammatic languages

Cubicon — Visual, Object-oriented, Commercial.
VPL-II. Language Classifications
A. Paradigms
8. Multi-paradigm languages
B. Visual representations
1. Diagrammatic languages
VPL-III. Language Features
A. Abstraction
1. Data abstraction
B. Control flow
C. Data types and structures
E. Event handling
VPL-V. Language Purpose
A. General-purpose languages

Dataflo MP — Commercial.

Design/CPN — Visual, Object-oriented, Commercial.
VPL-II. Language Classifications
B. Visual representations
1. Diagrammatic languages
VPL-V. Language Purpose
D. Scientific visualization languages

DT-VEE — Visual, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
    - 9. Object oriented languages
  - B. Visual representations
    - 1. Diagrammatic languages

Dylan — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 5. Functional languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
    - 2. Procedural abstraction
  - B. Control flow
  - E. Event handling
  - C. Data types and structures
- VPL-V. Language Purpose
  - A. General-purpose languages

Escalante — Visual.
- VPL-I. Environments and tools for VPLs
- VPL-II. Language Classifications
  - B. Visual representations
    - 1. Diagrammatic languages
    - 2. Iconic languages

Fabrik — Visual, Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
  - B. Visual representations
    - 1. Diagrammatic languages

Forms/3 — Visual, Object-oriented.
- VPL-II. Language Classifications
  - A. Paradigms
    - 4. Form-based and spreadsheet-based
    - 4. Form-based and spreadsheet-based languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - E. Event handling
  - F. Exception handling
- VPL-V. Language Purpose
  - A. General-purpose languages

ForShow — Commercial.

Goofy — Visual.
- VPL-I. Environments and tools for VPLs

Graqula — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
- VPL-V. Language Purpose
  - B. Db languages
  - C. Image-processing languages

GROOVE — Visual, Object-oriented.
- VPL-I. Environments and tools for VPLs
- VPL-V. Language Purpose
  - D. Scientific visualization languages

HI Graphs — Visual, Object-oriented.
- VPL-I. Environments and tools for VPLs

HI-VISUAL — Visual, Object-oriented.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
  - B. Visual representations
    - 2. Iconic languages
- VPL-V. Language Purpose
  - C. Image-processing languages

HSC InterActive — Commercial.

Hyperpascal — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures

HyperTalk — Visual, Object-oriented, Text based, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
    - 9. Object oriented languages
- VPL-III. Language Features
  - B. Control flow
  - E. Event handling
- VPL-V. Language Purpose
  - A. General-purpose languages
  - B. Db languages
  - E. User-interface generation languages

Iconauthor — Visual, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
  - B. Visual representations
    - 2. Iconic languages
- VPL-III. Language Features
  - B. Control flow
  - E. Event handling
- VPL-V. Language Purpose
  - A. General-purpose languages

Iconicode — Visual, Commercial.

LabView — Visual, Object-oriented, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
    - 9. Object oriented languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
    - 2. Procedural abstraction
  - B. Control flow
- VPL-V. Language Purpose
  - A. General-purpose languages

Layout — Visual, Object-oriented, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
  - B. Visual representations
    - 2. Iconic languages
- VPL-III. Language Features
  - B. Control flow
  - E. Event handling
- VPL-V. Language Purpose
  - A. General-purpose languages

LISP — Object-oriented, Text based, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 5. Functional languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
    - 2. Procedural abstraction
  - C. Data types and structures
- VPL-V. Language Purpose
  - A. General-purpose languages

MFL — Visual, Object-oriented.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
  - B. Visual representations
    - 1. Diagrammatic languages

Miro — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 11. Rule-based languages
  - B. Visual representations
    - 1. Diagrammatic languages

Modula-3 — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
- VPL-V. Language Purpose
  - A. General-purpose languages

Mondrian — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 10. Programming-by-demonstration languages
  - B. Visual representations
    - 3. Languages based on static pictorial sequences

MViews — Visual, Object-oriented, Text based.
- VPL-I. Environments and tools for VPLs
- VPL-II. Language Classifications
  - A. Paradigms
    - 5. Functional languages
  - B. Visual representations
    - 1. Diagrammatic languages

Newspeak — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
    - 9. Object oriented languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-V. Language Purpose
  - A. General-purpose languages

NoPumpG — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 4. Form-based and spreadsheet-based languages

O'Small — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures

Obliq — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
- VPL-V. Language Purpose
  - A. General-purpose languages

PC-TILES — Visual, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - B. Control flow
- VPL-V. Language Purpose
  - A. General-purpose languages

PhoneOne — Visual, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
  - B. Visual representations
    - 2. Iconic languages

PhonePro — Visual, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
  - B. Visual representations
    - 2. Iconic languages

Pict — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
  - B. Visual representations
    - 2. Iconic languages
- VPL-III. Language Features
  - B. Control flow

POLKA — Text based.
- VPL-I. Environments and tools for VPLs
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - B. Control flow
  - E. Event handling
- VPL-V. Language Purpose
  - D. Scientific visualization languages

PostScript — Object-oriented, Text based, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - B. Control flow
- VPL-V. Language Purpose
  - C. Image-processing languages
  - E. User-interface generation languages

Programming by Rehearsal — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 10. Programming-by-demonstration languages
  - B. Visual representations
    - 1. Diagrammatic languages

Prograph — Visual, Object-oriented, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
  - B. Visual representations
    - 2. Iconic languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
- VPL-V. Language Purpose
  - A. General-purpose languages

Pygmailion — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - B. Control flow
- VPL-V. Language Purpose
  - A. General-purpose languages

Q — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - B. Control flow
  - C. Data types and structures
- VPL-V. Language Purpose
  - A. General-purpose languages

RAPIDE — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures

SCHEME — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 5. Functional languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures

SELF — Visual, Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
    - 9. Object oriented languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
    - 2. Procedural abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
  - F. Exception handling
- VPL-V. Language Purpose
  - A. General-purpose languages

Show & Tell — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
  - B. Visual representations
    - 2. Iconic languages

Smalltalk — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 5. Functional languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
  - F. Exception handling
- VPL-V. Language Purpose
  - A. General-purpose languages

Snart — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 11. Rule-based languages
    - 9. Object oriented languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction

SPE — Visual, Object-oriented, Text based.
- VPL-I. Environments and tools for VPLs

SunPICT — Visual, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - B. Control flow
- VPL-V. Language Purpose
  - A. General-purpose languages

SysRPL — Object-oriented, Text based, Commercial.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
    - 9. Object oriented languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
    - 2. Procedural abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
- VPL-V. Language Purpose
  - A. General-purpose languages

tcl/tk — Object-oriented, Text based, Commercial.
- VPL-I. Environments and tools for VPLs
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
  - F. Exception handling
- VPL-V. Language Purpose
  - E. User-interface generation languages
  - A. General-purpose languages

ThingLab — Visual, Object-oriented.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
  - B. Visual representations
    - 2. Iconic languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures

Tinkertoy — Visual, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 5. Functional languages
  - B. Visual representations
    - 2. Iconic languages
- VPL-V. Language Purpose
  - A. General-purpose languages

Views — Visual, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
    - 11. Rule-based languages
- VPL-III. Language Features
  - B. Control flow
- VPL-V. Language Purpose
  - A. General-purpose languages

VIPR — Visual, Object-oriented.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
    - 9. Object oriented languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - B. Control flow
  - C. Data types and structures

VisaVis — Visual.
- VPL-II. Language Classifications
  - A. Paradigms
    - 5. Functional languages
- VPL-V. Language Purpose
  - A. General-purpose languages

Vista — Visual, Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 3. Data-flow languages
    - 6. Imperative languages
    - 8. Multi-paradigm languages
    - 9. Object oriented languages
  - B. Visual representations
    - 1. Diagrammatic languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures
  - E. Event handling
  - F. Exception handling
- VPL-V. Language Purpose
  - A. General-purpose languages

Visual Age — Commercial.

Visual AppBuilder — Visual, Object-oriented, Commercial.
- VPL-I. Environments and tools for VPLs
- VPL-V. Language Purpose
  - E. User-interface generation languages

Visual Smalltalk — Object-oriented, Text based, Commercial.
- VPL-I. Environments and tools for VPLs
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-V. Language Purpose
  - A. General-purpose languages

VS Enterprise — Commercial.
- VPL-I. Environments and tools for VPLs

ZZ — Object-oriented, Text based.
- VPL-II. Language Classifications
  - A. Paradigms
    - 6. Imperative languages
- VPL-III. Language Features
  - A. Abstraction
    - 1. Data abstraction
  - B. Control flow
  - C. Data types and structures
- VPL-V. Language Purpose
  - A. General-purpose languages

References

A Bibliography with Abstracts and Annotations

The following list contains references to a wide variety of printed resource information. Some of it describes the newest and the brightest of the development efforts; much of it is invaluable for the historic perspective and understanding it offers.

Where possible we have used authors' descriptions in their own words; note that not all of the authors are native English speakers. When no abstract was available, the authors of this Final Report have included a summary of the information available. Annotations are spread throughout the document.

This is a list to be used, annotated by the reader, and considered a solid coverage of the available information. The work of many talented authors is contained within.

[Abadi89] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin, *Dynamic Typing in a Statically Typed Language*, 1989.

Statically typed programming languages allow earlier error checking, better enforcement of disciplined programming styles, and generation of more efficient object code than languages where all type consistency checks are performed at run time. However, even in statically typed languages, there is often the need to deal with data whose type cannot be determined at compile time. To handle such situations safely, we propose to add a type Dynamic whose values are pairs of a value v and a type tag T where v has the type denoted by T. Instances of Dynamic are built with an explicit tagging construct and inspected with a type safe typecase construct.

This paper explores the syntax, operational semantics, and denotational semantics of a simple language including the type Dynamic. We give examples of how dynamically typed values can be used in programming. Then we discuss an operational semantics for our language and obtain a soundness theorem. We present two formulations of the denotational semantics of this language and relate them to the operational semantics. Finally, we consider the implications of polymorphism and some implementation issues.

[Adobe90] Adobe Systems Incorporated, **PostScript Language Reference Manual, 2nd edition**, 1990.

This book is the official reference to the PostScript language. It offers comprehensive coverage of the entire language, including the recent advancements of PostScript Level 2.

[Ae86] Tadashi Ae, Masafumi Yamashite, Wagner Chiepa Cunha, and Hiroshi Matsumoto , "Visual User-Interface of a Programming System MOPS^2", in *IEEE 1986 Workshop on VIsual Languages*, 1986, pgs 44–53.

This paper describes the visual aspects of a software development system MOPS^2, which provides an advanced environment for developing and testing of an embedded computer software. The software development on MOPS^2 is supported by modular programming and visual testing. Especially, the colored Petri net plays an important role for compact display on a conventional color graphics terminal.

[Agesen95] Ole Agesen, Lars Bak,Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar and Mario Wolczkox, *The SELF 4.0 Programmer's Reference Manual*, 1995, pgs 1–104.

Manual, covering language reference, Self world, guide to programming style, virtual machine reference

[Agha93] Gul Agha, Paul Wegner, and Akinori Yonezawa , **Research Directions in Concurrent Object Oriented Programming**, 1993.

[Agui84] T. Agui, Y. Arai, and M. Nakajima, "A Color Compression Algorithm for Natural Scenes and Animation Pictures", in *IEEE 1984 Workshop on Visual Languages*, 1984, pg 2.

In this paper, we propose a new color compression algorithm making use of linear division of color spaces; the division is in accordance with the value distribution in the color space.

The algorithm consists of four proessing steps and deals with raster scanned (r,g,b) values which are accessible sequentially.

This method makes the effectieness and efficiency of color look-up table higher when natural scene images or computer animation cel images are displayed.

Some results and evaluation of applying the algorithm are also described.

[Aho86] Alfred V. Aho,Ravi Sethi, and Jeffrey D. Ullman, **Compilers. Principles, Techniques, and Tools**, 1986.

This book is a descendant of Principles of Compiler Design by Alfred V. Aho and Jeffrey D. Ullman. Like its ancestor, it is intended as a text for a first course in compiler design. The emphasis is on solving problems universally encountered in designing a language translator, regardless of the source or target machine.

It also has a great cover.

[Alexandrov91] V.V. Alexandrov and N.D. Gorsky, **From Humans to Computers—Cognition Through Visual Perception**, 1991.

[Apple92] Apple Computer Eastern Research & Technology, **Dylan. An object-oriented dynamic language**, 1992.

Dylan is a new Object Oriented Dynamic Language (OODL) developed by the Eastern Research and Technology Lab of Apple Computer. Dylan was designed to make the advantages of OODLs available for commercial programming on a variety of computing devices. Dylan most closely resembles CLOS and Scheme. Other languages which influences the design of Dylan include Smalltalk, Self, and OakLisp.

[Apte93] Ajay Apte and Takayuki Dan Kimura , "A Comparison Study of the Pen and the Mouse in Editing Graphic Diagrams", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 352–357.

We report the results of an experiment comparing the merits of the pen and the mouse as drawing devices. For this study a pen-based graphic diagram editor equipped with a shape recognition algorithm was developed on GO's PenPoint operating system. A commercially available drawing program on NeXT was used for mouse-based editing. Twelve CS students were chosen as subjects and asked to draw four different diagrams of similar complexity: two with a pen and the other two with a mouse. The diagrams were chosen from the categories of dataflow visual language, Petri nets, flowcharts, and state diagrams. The results indicate that drawing by pen is twice as fast as drawing by mouse.

[Archibald92] Jerry L. Archibald, "The Economics of Software Reuse", in *Addendum to the Proceedings OOPSLA 91*, 1992.

In this note, we provide additional material to summarize and complement the discussion of the position papers presented at OOPSLA '91 in Phoenix.

Topic: The economics of Software Reuse. ...There was almost uniform concensus from the various speakers: the key issues we face in introducing effective reuse were economic, managerial, and social, and not only (or even predominantly) technical.

[Baecker90] Ronald M. Baecker and Aaron Marcus, **Human Factors and Typography for More Readable Programs**, 1990.

Program appearance has changed little since the first high-level languages were developed. While millions of people are writing programs, many of them are also reading programs, and this need to read the programs has received far less consideration than the need to write them. Attention has been focused on the logic of programming languages but not sufficiently on their visual presentation; tools have been built to facilitate program composition and editing, but not program perusal or understanding. Reading this book could change some of that. Productivity is enhanced when the program is easier to read. The authors provide an introduction to the issues, methods and results of effective program presentation. A good book.

[Baecker91] Ronald M. Baecker, Ian Small, and Richard Mander, "Bringing Icons to Life", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pgs 1–6.

Icons are used increasingly in interfaces because they are compact "universal" pictographic representations of computer functionality and processing. Animated icons can bring to life symbols representing complete applications or functions within an application, thereby clarifying their meaning, demonstrating their capabilities, and even explaining their method of use. To test this hypothesis, we carried out an iterative design of a set of animated painting icons that appear in e HyperCard tool palette. The design discipline restricted the animations to 10 to 20 second sequences of 22x20 pixel bit maps. User testing was carried out on two interfaces—one with the static icons, one with the animated icons. The results showed significant benefit from the animations in clarifying the purpose and functionality of the icons.

[Barfield91] Lon Barfield, Eddy Boeve, and Steven Pemberton , "The Views User-Interface System", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 415.

Views is a user-interface system where the user interface is a layer above applications, guaranteeing consistency of interface, and with a data-layer implementing external object representation, allowing exchange of objects between applications without loss of structure. The user's model is that all actions are achieved by editing documents that describe the world; an invariant or constraint-like system assures that the world is brought up-to-date with the description. Among the innovative aspects, object presentation is independent of the content of objects, so that objects can be viewed in different ways simultaneously, and presentations can be easily changed, even on-the-fly.

[Barnes94] J.G.P. Barnes, **Programming in Ada (Plus an Overview of Ada 9X), 4th edition**, 1994.

Written by a key member of the original Ada design team, this book is acknowledged as the definitive text and reference for Ada programmers and students. This fourth edition, remaining focused on the current ANSI 83 standard, reflects the imminent Ada 9x standard. All features of Ada that will be affected by the new version are highlighted and design rationale described in detail; details of the syntax changes; and a n Ada 9x tutorial.

[Bauersfeld91] Penny F. Bauersfeld and Jodi L. Slater, "User-Oriented Color Interface Design: Direct Manipulation of Color in Context", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 417.

Color functionality for personal computers is currently limited and does not adequately address user needs. By introducing principles of color theory and design to interface tools we can provide users with innovative ways to manipulate color. This paper describes color tools based on a uniform perceptual color space that account for the relativity of color. Color interface ideas based on such user-oriented color models and approaches are suggested.

[Baumgartner90] Gerald Baumgartner and Ryan Stansifer, "A Proposal to Study Type Systems for Computer Algebra", in *RISC [Research Institute for Symbolic Computation] LINZ Series no. 90-07.0*, 1990.

It is widely recognized that programming languages should offer features to help structure programs. To achieve this goal, languages like Ada, Modula-2, object-oriented languages, and functional languages have been developed. The structuring techniques available so far (like modules, classes, parametric polymorphism) are still not enough or not appropriate for some application areas. In symbolic computation, in particular computer algebra, several problems occur that are difficult to handle with any existing programming language. Indeed, nearly all available computer algebra systems suffer from the fact that the underlying programming language imposes too many restrictions.

We propose to develop a language that combines the essential features from functional languages, object-oriented languages, and computer algebra system in a semantically clean manner. Although intended for use in symbolic computation, this language should prove interesting as a general purpose programming language. The main innovation will be the application of sophisticated type systems to the needs of computer algebra systems. We will demonstrate the capabilities of the language by using it to implement a small computer algebra library. This implementation will be compared against a straightforward Lisp implementation and against existing computer algebra systems. Our development should have an impact both on the programming languages world and on the computer algebra world.

[Baumgartner94] Gerald Baumgartner and Vincent F. Russo, *Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism*, 1994.

C++ uses inheritance as a substitute for subtype polymorphism. We give examples where this makes the type system too inflexible. We then describe a conservative language extension that allows us to define an abstract type hierarchy independent of any implementation hierarchies, to retroactively abstract over an implementation, and to decouple subtyping from inheritance.This extension gives the user more of the flexibility of dynamic typing while retaining the efficiency and security of static typing. With default implementations and views we provide flexible mechanisms for implementing an abstract type by different concrete class types. We first show how our language extension can be implemented in a preprocessor to a C++ compiler, and then detail and analyze the efficiency of an implementation we directly incorporated in the GNU C++ compiler.

[BaumgartnerXX] Gerald Baumgartner and Vincent F. Russo, *Implementing Signatures for C++* , XX.

In this paper we overview the design and implementation of a language extension to C++ for abstracting types and for decoupling subtyping and inheritance. This extension gives the user more of the flexibility of dynamic typing while retaining the efficiency and security of static typing. We discuss the syntax and semantics of this language extension, show examples of its use, and present and analyze the cost of three different implementation techniques: a preprocessor to a C++ compiler, an implementation in the front end of a C++ compiler, and a low-level back-end based implementation.

[Beguelin92] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, et al., "HeNCE: graphical developement tools for network-based concurrent computing", in *Proceedings. Scalable High Performance Computing Comference SHPCC - 92*, 1992, pgs 129 – 136.

[Bell91] Brigham Bell, John Rieman,and Clayton Lewis, "Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pgs 7–12.

Traditional programming language design has focussed on efficiency and expressiveness, with minimal attention to the ease with which a programmer can translate task requirements into statements in the language, a characteristic we call "facility". The programming walkthrough is a method for assessing the facility of a language design before implementation. We describe the method and its predictions for a graphical programming language, ChemTrains. These predictions are contrasted with protocols of subjects attempting to write their first ChemTrains program. We conclude that the walkthrough is a valuable aid at the design stage, but it is not infallible. Our results also suggest that it may not be enough for programmers to know how to solve a problem; they must also understand why the solution will succeed.

[Bell92] Brigham Bell and Wayne Citrin, "Simulation of Communications Protocols through Graphical Transformation Rules", in *Proceedings of the International Workshop AVI '92*, 1992, pg 208.

We present a novel approach to the specification and simulation of communications protocols, through graphical transformation rules. These rules are expressed using ChemTrains, a graphical transformation system. For the protocols tested, the sets of rules are concise and easily constructed. Such a system should be suitable for rapid prototyping, performance analysis, and instruction. Whether executable protocol code can also be generated from the graphical transformation rules remains an open question.

[Bell93] Brigham Bell and Clayton Lewis, "ChemTrains: A LANGUAGE FOR CREATING BEHAVING PICTURES", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 188–195.

ChemTrains is a rule-based language in which both the condition and action of each rule are specified by pictures. A ChemTrains rule will execute when the topology rather than the geometry of a pattern matches a portion of the simulation picture, enabling rules to be drawn at a high level of abstraction. The language is independent of any specific application domain and is accessible to people with limited programming knowledge.

[Bell94] B. Bell , W. Citrin, C. Lewis, J. Rieman, R. Weaver, N. Wilde and B. Zorn, "Using the Programming Walkthrough to Aid in Programming Language Design", in *Software - Practice and Experience*, 1994, pgs 1 – 25.

The programming walkthrough is a method for assessing how easy or hard it will be for users to write programs in a programming language. It is intended to enable language designers to identify problems early in design and to help them choose among alternative designs. We describe the method and present experience in applying it in four language designs. We describe the method and present experience in applying it in four language design projects. Results indicate that the method is a useful supplement to existing design approaches.

[Bianchi93] N. Bianchi, P. Bottoni, P. Mussio, and M. Protti, "Cooperative Visual Environments for the Design of Effective Visual Systems", in *Journal of Visual Languages and Computing*, 1993, pgs 357–381.

This paper describes the architecture of cooperative visual enviroments (CVE). This proposal stems from the findings of several experiments which suggested overcoming the limitations of first-generation user-interface management systems (UIMS) by allowing the users to determine their own computational environment. To avoid user disorientation, as well as the possibility of creating ambiguous or contradictory systems, a novel discipline for the specification and use of the tools is adopted. A systemic approach has been proposed to identify the variables needed to use, observe and adapt a CVE. The design and implementation of tools satisfying this discipline led to the definition of network objects, generalizing composite objects, and to the introduction of typed links allowing a new technique for message passing. The paper illustrates the above points by discussing the rationale behind the design of CVEs, deriving the requirements which CVEs have to satisfy and outlining the architecture with the fundamental mechanisms which allow their use and evolution. The nature of the proposal is also clarified through an example drawn from a real case.

[Bier91] Eric A. Bier and Ken Pier, "Documents as User Interfaces", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 443.

Introduction: Each year the electronic documents community produces better tools for creating and changing document elements, including text, illustrations, video, and animation. At the same time, the user interface community works to build interfaces that improve the quality of interaction by effectively presenting information to the user and making it easy to act on and manipulate that information. These efforts can be combined by using documents as user interfaces. We have implemented an architecture, EmbeddedButtons, that allows arbitrary document elements, managed by arbitrary editors, to behave as buttons.

[Birrell94] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber, *Network Objects*, 1994.

A network object is an object whose methods can be invoked over a network. This report describes the design and implementation of a network objects system for Modula-3. The system is novel for its overall simplicity. The report includes a thorough description of realistic marshaling algorithms for network objects, precise informal specifications of the major internal interfaces, preliminary experience, and performance results.

[Borges90] Jose A. Borges and Ralph E. Johnson, "Multiparadigm Visual Programming Language", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 233–240.

Visual programming languages have not been successful for general purpose programming. This paper argues that multiparadigm visual programming languages will be better suited for general purpose programming than languages based on a single paradigm. It illustrates these points with the Visual ToolSet, a multiparadigm visual programming language.

[Borschen94] Christoph Boschen, Christian Fecht, Andreas V. Hense, and Reinhard Wilhelm, *An Abstract Machine for an Object-Oriented Language with Top-Level Classes*, 1994.

Object-oriented programming languages where classes are top-level, ie, not first-class citizens, are better suited for compilation than completely dynamic languages like Smalltalk or Self. In O'small, a language with top-level classes, the compiler can statically determine the inheritance hierarchy. Due to late binding, the class of the receiver of a message must be determined at run time. After that a direct jump to the corresponding method is possible. Method lookup can thus be done in constant time.

We present an abstract machine for O'small based on these principles. It is a concise description of a portable O'small implementation.

[BothnerXXa] Per Bothner, *Q, The Q Language*, XX.

no abstract provided. Table of Contents includes 1) invoking q, 2) syntax, 3) control structures, 4) numbers, 5) symbols, 6) mappings, 7) sequences, 8) atoms, 9) union objects, 10) declarations, 11) functions, 12) operators, 13) streams, 14) records, 15) assignment, 16) system interface, 17) lisp support.

[BothnerXXb] Per Bothner, *Q Shell A Programming-Language Shell*, XX.

Modern high-level programming languages provide multiple data types (such as numbers, strings, and lists), as well as first-class function values. Most utility languages (awk, perl, tcl) and most shells only have one (or a few) data types (strings), but they are very convenient for manipulating text or invoking programs. This paper discusses the issues involved in getting the best of both worlds, in the context of the Q programming language.

For example, executing a program has the same syntax as a function call, a pipe is function composition, and a disk file just a persistent string. Similarly, a disk file just a persistent string. These goals have implications for the syntax and the control structure of the programming language.

[Brade92] Kathleen Brade, Mark Guzdial , Mark Steckel, and Elliot Soloway, "Whorf: A Visualization Tool for Software Maintenance", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 148.

Software maintenance programmers face the daunting task of understanding and modifying complex, unfamiliar programs that contain delocalized plans (conceptually related code that isn't located contiguously in a program). Our research shows that programmers use an as-needed strategy when searching for the delocalized components which they need to understand. We have developed a maintenance tool, Whorf, that provides explicit support for visualizing and understanding delocalized plans using as as-needed strategy. Whorf supports this strategy through multiple, concurrent views of the software with instant, easy access to additional views. It supports understanding of delocalized plans by providing hypertext links between views to highlight interactions between physically disparate components. A study comparing the usage of Whorf and paper documentation shows that the dynamic views and structure supported by Whorf provide information more quickly and easily than the static structure of paper documentation.

[Brown93] Marc H. Brown, *The 1992 SRC Algorithm Animation Festival*, 1993.

During the last two weeks of July 1992, twenty researchers at Digital Equipment Corporation's Systems Research Center participated in the 1st Annual SRC Algorithm Animation Festival. Only two of the researchers had previously animated an algorithm, and not too many more had ever written an application that involved graphics. In this paper, we report on the Animation Festival, describing why we did it and what we did, and commenting on what we learned.

[Brown94] Marc H. Brown, *The 1993 SRC Algorithm Animation Festival*, 1994.

This report describes the 1993 SRC Algorithm Animation Festival. The festival continues an experiment in developing algorithm animations by non-experts, started the previous year, and described in SRC Research Report #98. This year nineteen researchers at Digital Equipment Corporation's systems Research Center worked for two weeks on animating algorithms. Most of the participants had little (if any) experience writing programs that involved graphics. This report explains why we organized the festival, and describes the logistics of the festival and the advances in our algorithm animation system. This report presents the complete code for a simple, but non-trivial, animation of first-fit binpacking. Finally, this report contains snapshotsfrom the animations produced during the festival.

[Bryson93] Steve Bryson and Steve Feiner, *IEEE Symposium on Research Frontiers in Virtual Reality*, 1993.

[Burdea93] Grigore Burdea and Philippe Coiffet, **Virtual Reality Technology**, 1993.

[Burnett90] Margaret M. Burnett and Allen L. Ambler , "Efficiency Issues in a Class of Visual Language", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 209–214.

This paper identifies a class of visual programming languages whose members share a common group of underlying principles. In this class, the demand-driven temporal-assignment visual language model, the elimination of certain duplicate computations is a natural by-product of the model. The potential time and space complexity characteristics for visual languages based upon this model are discussed, and a method that makes use of these characteristics is presented.

[Burnett91] Margaret M. Burnett, **Abstraction in the Demand-Driven Temporal-Assignment, Visual Language Model**, 1991.

Ph.D. thesis. Includes a very extensive annotated bibliography.

[Burnett92a] Margaret M. Burnett and Allen L. Ambler , "Generalizing Event Detection and Response in Visual Programming Languages", in *Proceedings of the International Workshop AVI '92*, 1992, pg 334.

Although direct interaction plays an important role in visual programming languages (VPLs), most approaches to events in VPLs treat event-handling as a side issue, often via special-purpose libraries or tools. This paper presents a general approach to events in VPLs which distinguishes between event-detection and event-response. Event-detection is treated as an abstraction, resulting in data compatible and composable with other calculations and data. This approach is extremely flexible, allowing the user to define event responses to virtually any event sequence.

[Burnett92b] Margaret M. Burnett and Allen L. Ambler , "A Declarative Approach to Event-Handling in Visual Programming Languages", in *1992 IEEE Workshop on Visual Languages*, 1992, pgs 34–40.

In this paper, we address the question of event-handling for declarative visual languages. In the approach presented, system-level, interactive, and user-defined events are fully supported, while still maintaining the property of referential transparency. An approach to time termed temporal assignment provides a unifying mechanism for events to be defined as ordinary sequences of values, and conversely for ordinary sequences of values to be defined as events. This allows event-handling without additional concepts, and in particular provides a natural means for the user to define higher-level events of any kind.

[Burnett92c] Margaret M. Burnett and Allen L. Ambler , "Declarative Approach to Event-Handling in Visual Programming Languages", in *1992 IEEE Workshop on Visual Languages*, 1992, pgs 34–40.

In this paper we address the question of event-handling for declarative visual languages. In the approach presented, system-level, interactive, and user-defined events are fully-supported, while still maintaining the property of referential transparency. An approach to time termed temporal assignment provides a unifying mechanism for events to be defined as ordinary sequences of values, and conversely for ordinary sequences of values to be defined as events. This allows event-handling without additional concepts, and in particular provides a natural means for the user to define higher-level events of any kind.

[Burnett93] Margaret M. Burnett, "Types and Type Inference in a Visual Programming Language", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 238–243.

In this paper, the uses of types and type inference in visual languages are explored. First, we discuss how the goals of a type system must differ for visual languages from those of a type system for textual languages. We then present a type system developed under these goals for the visual language Forms/3. Within the context of this system, issues of particular importance in visual languages are examined, including maintaining the user"s conceptual model, the avoidance of language restrictions solely to support a type system, and how the visual process of programming can provide additional information to the type system.

[Burnett94a] Margaret Burnett and Benjamin Summers , *Some Real-World Uses of Visual Programming Systems*, 1994.

What kinds of practical uses are people making of visual programming today in the real world? To find this out, we gathered information about users of visual programming systems. The information came in response to a newsgroup posting asking people to report their uses of visual programming.... Table 1 contains information about these users, which systems they use, and whether they use it for general purpose programming or for a domain-specific purpose intended by the system they are using. ... Table 2 presents the specific uses being made of these visual programming systems.

[Burnett94b] Margaret M. Burnett and Marla J. Baker, *A Classification System for Visual Programming Languages*, 1994.

We have developed a classification scheme for classifying visual programming language research papers. This paper presents the scheme, the motivations for developing it, and examples of its use.

[Burnett94c] Margaret M. Burnett and Allen L. Ambler , "Declarative Visual Languages", in *Journal of Visual Languages and Computing*, 1994, pgs 1–3.

A commentary and overview by Burnett and Ambler as Guest Editors of JVLC on the topic of declarative visual languages.

[Burnett94d] Margaret M. Burnett and Allen L. Ambler , "Interactive Visual Data Abstraction in a Declarative Visual Programming Language", in *Journal of Visual Languages and Computing*, 1994, pgs 29 – 60.

Visual data abstraction is the concept of data abstraction for visual languages. In this paper, first we discuss how the requirements of data abstraction for visual languages differ from the requirements for traditional textual languages. We then present a declarative approach to visual data abstraction in the language Forms/3. Within the context of this system, issues of particular importance to declarative visual languages are examined. These issues include enforcing information hiding through visual techniques, supporting abstraction while preserving concreteness, conceptual simplicity, and specification of a type's appearance and interactive behavior as part of its definition. Interactive behavior is seen to be part of the larger problem of event-handling in a declarative language. A significant feature is that all programming and execution are done in a fully-integrated visual manner, without requiring other languages ortools for any part of the programming process.

[Burnett94d] Margaret M. Burnett and Allen L. Ambler , "Interactive Visual Data Abstraction in a Declarative Visual Programming Language", in *Journal of Visual Languages and Computing*, 1994, pgs 29 – 60.

Visual data abstraction is the concept of data abstraction for visual languages. In this paper, first we discuss how the requirements of data abstraction for visual languages differ from the requirements for traditional textual languages. We then present a declarative approach to visual data abstraction in the language Forms/3. Within the context of this system, issues of particular importance to declarative visual languages are examined. These issues include enforcing information hiding through visual techniques, supporting abstraction while preserving concreteness, conceptual simplicity, and specification of a type's appearance and interactive behavior as part of its definition. Interactive behavior is seen to be part of the larger problem of event-handling in a declarative language. A significant feature is that all programming and execution are done in a fully-integrated visual manner, without requiring other languages or tools for any part of the programming process.

[Burnett95] Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, **Visual Object Oriented Programming**, 1995.

This first book on the union of two rapidly growing approaches to programming —visual programming and object technology—provides a window on a subject of increasing commercial importance. (t is an introduction and reference for cutting-edge developers, and for researchers, students, and enthusiasts interested in the design of visual OOP languages and environments.

[Card91] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay, "The Information Visualizer, an Information Workspace", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 181.

This paper proposes a concept for the user interface of information retrieval systems callled an information workspace. The concept goes beyond the usual notion of an information retrieval system to encompass the cost structure of information from secondary storage to immediate use. As an implementation of the concept, the paper describes an experimental system, called the Information Visualizer, and its rationale. The system is based on (1) the use of 3D/Rooms for increasing the capacity of immediate storage available to the user (2) the Cognitive Co-processor scheduler-based user interface interaction architecture for coupling the user in information agents, and (3) the use of information visualization for interacting with information structure.

[Cardelli94] Luca Cardelli, *A Language with Distributed Scope*, 1994, pgs 1–15.

Obliq is a lexically-scoped, untyped, interpreted language that supports distributed object-oriented computation. Obliq objects have state and are local to a site. Obliq computations can roam over the network, while maintaining network connections. Distributed lexical scoping is the key mechanism for managing distributed computations.

[Cardelli95] Luca Cardelli, *Obliq. A Language with Distributed Scope*, 1995.

Obliq is lexically-scoped, untyped, interpreted language that supports distributed object-oriented computation. Obliq objects have state and are local to a site. Obliq computations can roam over the network, while maintaining network connections. Distributed lexical scoping is the key mechanism for managing distributed computations.

[Catarci92c] Tiziana Catarci, Marie Francesca Costabile, and Stefano Levialdi, *Advanced Visual Interfaces*, 1992.

Proceedings of the International Workshop AVI '92, Rome, Italy, May 27-29, 1992

[Catteneo86] G. Cattaneo, A. Guercioi, S. Levialdi, and G. Tortora, "IconLisp: An Example of a Visual Programming Language", in *IEEE 1986 Workshop on VIsual Languages*, 1986, pgs 22–25.

This paper presents a visual extension of an existing funcitonal programming language (LISP) so as to implement visual programming. Two advantages may be gained: the first one leading to an interactive teaching of LISP properties and behavior and to the possibility of writing correct programs using icons by means of visual feedback.

A scenario is also included.

[Chambers91] Craig Chambers and David Ungar, *Making Pure Object-Oriented Languages Practical*, 1991.

In the past, object-oriented language designers and programmers have been forced to choose between pure message passing and performance. Last year, our SELF system achieved close to half the speed of optimized C but suffered from impractically long compile times. Two new optimization techniques, deferred compilation of uncommon cases and non-backtracking splitting using path objects, have improved compilation speed by more than an order of magnitude. SELF now compiles about half as fast as an optimizing C compiler and runs at over half the speed of optimized C. This new level of performance may make pure object-oriented languages practical.

[Chang86] S.-K. Chang, T. Ichikawa, and P.A. Ligomenides, **Visual Languages**, 1986.

Edited by some of the most preeminent names in the field, this collection of papers on visual languages offers a wide perspective of the topic.

[Chang90] Shi-Kuo Chang, *1990 IEEE Workshop on Visual Languages*, 1990.

1990 IEEE Workshop on Visual Languages, October 4-6, 1990, Skokie, IL, USA

[Citrin] Wayne Citrin, Michael Doherty, and Benjamin Zorn , *A Formal Definition of Control Semantics in a Completely Visual Language.*

Visual representations of programs can facilitate program understanding by presenting aspects of programs using explicit and intuitive representations. To explore this idea, we have designed a completely visual static and dynamic presentation of an imperative programming language. Because our representation of control is completely visual, programmers using this language can understand the static and dynamic semantics of programs using the same framework. In this paper, we describe the semantics of our language, both informally and formally, focusing on support for control constructs. We also prove that using our language to model common high-level constructs is semantically sound. This paper is an expanded version of [5]; we present a revised and more complete treatment of the language's graphical semantics, introduce the concept of canonical configurations, and discuss the representation of function return values.

[Citrin93] Wayne V. Citrin , "Requirements for Graphical Front Ends for Visual Languages", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 142–150.

Although great progress has been made in the development of visual languages, little attention has been paid to how diagrams in such languages should be entered into computers. This issue will have a great impact, however, on users' acceptance of visual languages. We present an analysis of the features of visual languages influencing diagram entry, and also the results of an experiment comparing users' performance on different types of diagram editors. This data is then used to suggest design guidelines for usable visual language front ends.

[Citrin93] Wayne Citrin, Michael Doherty, and Benjamin Zorn, *Control Constructs in a Completely Visual Imperative Programming Language*, 93.

Visual representations of programs can facilitate program understanding by presenting aspects of programs using explicit and intuitive representations. We have designed a completely visual static and dynamic representation of an imperative programming language. Because our representation of control is completely visual, programmers of this language can understand the static and dynamic semantics of programs using the same framework. In this paper, we describe the semantics of our language, both informally and formally, focusing on support for control constructs. We also illustrated how simple programs written in this language will look both statically and dynamically. Our representation makes explicit some parts of program execution that are implicit in textual representations, thus our programs may be easier to understand.

[Coad93] Peter Coad and Jill Nicola, **Object-Oriented Programming**, 1993.

A good book on object oriented programming. A very thorough, well presented discussion.

[Collins95] Dave Collins , **Designing Object-Oriented User Interfaces**, 1995.

An interesting, useful book on human interface design for object oriented systems. A hands-on kind of book.

[Costagliola90] Gennaro Costagliola and Shi-Kuo Chang, "DR PARSERS: a generalization of LR parsers", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 174–180.

In this paper we will present a way to construct a parser for a visual language whose specification can be done by acontext-free grammar. The main idea is to allow a traditional LR parser to choose the next symbol to parse from a two-dimensional space. For this purpose an intermediate representation of the picture is needed. We will then make use of iconic indices. Cases of ambiguity are analyzed and some ways to avoid them are presented. Moreover the results of a practical implementation using the Yacc tool have been given.

[Cota90a] Bruce A. Cota and Robert G. Sargent, *Simulation Algorithms for Control Flow Graph Models*, 1990.

We discuss algorithms for simulation based on control flow graph models, which were introduced in an earlier paper. We first review control flow graphs and the "basic algorithm" for simulation based on them. We then review the fact that the basic algorithm automatically computes a form of lookahead without any null message passing or optimistic assumptions, but that some additional mechanism is required to avoid deadlock. We discuss five different versions of this algorithm, each of which uses a different means of avoiding deadlock — null message passing, deadlock detection and resolution, optimistic computation, a mixed conservative/optimistic algorithm, and an asynchronous sequential algorithm. In the null message passing algorithm, the lookahead automatically computed by the basic algorithm is used to reduce the number of null messages sent. In the optimistic algorithm, this same lookahead is used to avoid unnecessary roll backs. In the mixed conservative/optimistic approach, the lookahead is used as the basis for a conservative computation "underlying" an optimistic computation. The optimistic computation is then used only when conservative computation alone is not sufficient to keep all available processors busy. In the asynchronous sequential approach, the lookahead is used to avoid some event list operations. All of these algorithms use only the model representation and do not require any additional information about the model from the modeler.

[Cota90b] Bruce A. Cota and Robert G. Sargent, *Control Flow Graphs: A Method of Model Representation for Parallel Discrete Event Simulation*, 1990.

"A model representation called "control flow graphs" is developed that makes useful information for parallel discrete event simulation explicit. An algorithm for parallel discrete event simulation that makes use of these properties is given and an algorithm for automatically computing null messages for a conservative parallel simulation is given The idea is that a user could use some language based on this representation to describe a model, and an efficient parallel simulation could be carried out based on that model description, without requiring the user to explicitly program additional knowledge about the model into the simulation."

[Coulmann93] Ludwig Coulmann, "General Requirements for a Program Visualization Tool to be used in engineering of 4GL programs", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 37–41.

Program visualization can be used profitably to help a programmer gain an understanding of the program's meaning. In our context this process is called program analysis. The paper first points out that program analysis is highly individual and is influenced by the person involved and by the aim of the process. Secondly, it describes what consequences evolve out of the program analysis characteristics for a supporting tool. Four distinct, general properties of a visualization tool are presented, emphasizing the user and his changing interests. Thirdly, concepts are given for the visualization of 4GL-programs and a specific tool is decribed as an example of how the outlined requirements translate to a real application. A tree is used to represent different structural relations in the program and icons at the nodes facilitate the tree's perception.

[Coutaz91] Joelle Coutaz and Sandrine Balbo , "Applications: A Dimension Space for User Interface Management Systems", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 27.

This article presents an abstract space of dimensions which characterize the behavior of applications (i.e., functional cores) with regard to UIMS components. These dimensions such as responsiveness, accessibility, and instantiability, constitute a conceptual framework which captures the notion of functional core in terms adequate for UIMS designers. The dimension space may also be viewed as a requirements list for designing new UIMSs as well as set of criteria for evaluating UIMSs.

[Cox92] P.T. Cox and T. Pietrzykowski, "Visual Message Flow Language MFL and Its Interface", in *Proceedings of the International Workshop AVI '92*, 1992, pg 348.

The visual language MFL (Message Flow Language) is presented. MFL reverses the usual view of dataflow making the objects of a computation the most significant items, and giving them concrete representations related to their visualisations where appropriate. The interactions of these objects are described in terms of message flows routed by controls external to the objects. The visual syntax and the semantics of MFL are presented, based on an extensive example. Although MFL is a general purpose language for describing computations by connected devices, one of its prime application is the programming of visual interfaces, where the objects are interface items. Since the language depends very heavily on multiple visualisations of program elements, depending on context, the interface to the editor and debugger is most important. An overview of interface philosophy is presented.

[Cox92] Kenneth C. Cox and Gruia-Catalin Roman, "Abstraction in Algorithm Animation", in *1992 IEEE Workshop on Visual Languages*, 1992 , pg 18.

Abstraction of information into visual form plays a key role in the development of algorithm animations. We present a classification for abstraction as applied to algorithm animation. The classification emphasizes the expressive power of the abstraction, ranging from direct presentation of the program's state to complex animations intended to explain the behavior of the program. We illustrate our classification by presenting several visualizations of a shortest path algorithm.

[Cox94] Kenneth C. Cox and Grui-Catalin Roman , "A Characterization of the Computational Power of Rule-based Visualization", in *Journal of Visual Languages and Computing*, 1994, pgs 5 – 27.

Declarative visualization is a paradigm in which the process of visualization is treated as a mapping from some domain (typically a program) to an image. One means of declaring such mappings is through the use of rules which specify the relationship between the domain and the image. This paper examines the computational power of such rule-based mappings, focusing on their ability to construct the types of images typically desired in program visualization specifications.

[Daum94] Thorsten Daum, Douglas G. Fritz, and Robert G. Sargent, *A Graphical User Interface for Hierarchical Interconnection Graph Specification*, 1994.

A GUI is presented for the specification of Hierarchical Interconnection Graphs. Brief overviews of HI Graphs, how to specify the coupled component specifications of HI graphs, and how the GUI is designed and implemented are presented. A queuing model of a production system is used as an example in the overviews. The GUI was developed on a Unix workstation using the C++ programming language and the InterViews C++ graphical interface toolkit.

[deBakker90] J.W. de Bakker, W.P. de Roever, and G. Rozenberg, "Foundations of Object-Oriented Languages", in *Lecture Notes in Computer Science*, 1990.

Proceedings of the REX School/Workshop, Noordwijkerhout, The Netherlands, MayJune 1990.

[Del Bimbo93] Alberto Del Bimbo, Enrico Vicario, and Daniele Zingoni, "Visual Specifications of Virtual Worlds", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 376–378.

Verisimilitude to real world phenomena is the most appealing feature of Virtual Reality environments. A high degree of realism can be attained through a fine representation of the visual appearance of the actors playing in the virtual world, but a verisimilar replication of their behavior is necessary too. Preliminary results on a project aiming at the construction of a tool for the visual specification of the behavior pattern of virtual world actors are presented.

[DeTreville93] John DeTreville , "The GraphVBT Interface for Programming Algorithm Animations", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 26–31.

The GraphVBT interface is used for programming algorithm animations within the Zeus system. GraphVBT provides a small number of primitive object types—vertices, edges, vertex highlights, and polygons—and methods on them that are flexible enough to apply to animations of many classes of algorithms. The interface is presented and examples of its use are shown.

[Di Battista92] G. Di Battista, G. Liotta, M. Strani, and F. Vargiu, "Diagram Server", in *Proceedings of the International Workshop AVI '92*, 1992, pg 415.

A diagram is a drawing on the plane consisting of a set of symbols (nodes) and a set of connections (edges) between nodes.

Diagrams are widely used as an interaction language with users in a large set of applications like information system analysis and design (Entity-Relationship diagrams, Flow diagrams, Jackson diagrams), software engineering (Petri Nets, Subroutine call graphs, State transition diagrams), CAD/CAM applications and VLSI circuit layout.

As the number of objects (nodes and edges) involved in a diagram grows, the effort required to solve the conflicts of physical layout with the desired appearance becomes greater.

Having a tool that automatically manage graphs allows the user not to be involved with problems regarding graph drawing but to focus attention on the meaning of what the graph represents.

The problem of designing tools for automatically drawing diagrams has been intensively studied in the last years [offers references]; however, existing tools are not enough parametric and user-friendly both to generate a wide variety of diagrams and to be easily used by a not expert final user.

Diagram Server (DS in what follows) (offers references) is a network server designed to solve the above problems; it offers to its clients an effective set of facilities to easily represent and manipulate diagrams through a multiwindowing environment; moreover DS can be customized according to different application contexts and graphic environments.

[Di Gesu92] Vito DiGesu and Domenico Tegolo , "The Iconic Interface for the PIctorial C Language", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 119.

Iconic environments intend to provide expressive tools to implement, to debug and to execute programs. Moreover its pictorial constructs guide the user to design algorithms in an interactive fashion. Visual interfaces are especially required whenever programs run on an heterogeneous and reconfigurable multi-processor system oriented to image analysis. Pictorial tools help the user to control the scope of variables, and the distribution of the tasks into the processors. In this paper, the general design, the visual-syntax, and the implementation of the first prototype of an iconic user interface for the PIctorial C Language (PICL) are described.

[Dillon92] L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna, *A Graphical Interval Logic for Specifying Concurrent Systems*, 1992.

The paper describes a graphical interval logic that is the foundation of a toolset supporting formal specifications and verification of concurrent software systems. Experience has shown that most software engineers find standard temporal logic difficult to understand and to use. The objective of this work is to enable software engineers to specify and reason about temporal properties of concurrent systems more easily by providing them with a logic that has an intuitive graphical representation and with tools that support its use. To illustrate the use of the graph,ical logic, the paper provides some specifications for an elevator system and proves several properties of the specification. The paper also describes the toolset and the implementation.

[Dillon94] L.K. Dillon, G. Kutty, P.M. Melliar-Smith, L.E. Moser, and Y. S. Ramakrishna, "Visual Specifications for Temporal Reasoning", in *Journal of Visual Languages and Computing*, 1994, pgs 61–81.

Graphical Interval Logic (GIL) is a visual temporal logic in which formulas resemble the informal timing diagrams familiar to systems designers and software engineers. It provides an intuititve and natural visual notation in which to express specifications for concurrent system and retians the benefits of a formal notation. A visual editor permits GIL specifications to be easily constructed, and to be stored in and retrieved from files. The editor interfaces with a proof checker and model generator, which permit verification of temporal inferences. The paper shows how graphical specifications are created and used to reason about temporal properties of systems. It shows how pictures that formalize temporal arguments enhance understanding and help motivate successful proof strategies.

[Dony92] Christophe Dony, Jan Purchase, and Russel Winder, "Exception Handling in Object-Oriented Systems, Report on ECOOP '91 Workshop W4", in *OOPS Messenger*, 1992, pgs 17–30.

Introduction...

Today, object-oriented language designers and users are showing a renewed interest in exception handling. Exception handling systems have recently been, or are being, integrated into many object-oriented languages (C+ (Koenig & Stroustrup 1990), CommonLisp (+CLOS) (Pitman, 1988), Eiffel (Meyer, 1988) Smalltalk (ParcPlace Systems, 1989) etc.). There are two main reasons for this interest. Firstly, as object-oriented languages become more sophisticated, the problem of coping with exceptional situations occurring at run-time becomes more complex and the need for appropriate tools and language mechanisms to detect, handle and correct errors more crucial (Purchase & Winder, 1991). Secondly, the infeasibility of achieving information hiding and modularity at a large scale without exception handling system.

The goal of the workshop, organized by the authors, was to compare existing systems developed for object-oriented languages and to address various issues connected with the semantics and the implementation of these systems. The call for participation for the workshop stated many of these issues, the main ones being:

- the relationship between exception handling, software quality, modularity, and reusability.
- the problem of object consistency in the presence of exceptional events.
- the level of modularity used to associate handlers with code.
- the role of formal specifications.
- the use of reflection.
- the problems raised by concurrency.

Some other issues were raised during the workshop (continues on into document)

[Dow92] Chyi-Ren Dow, Mary Lou Soffa, and Shi-Kuo Chang , "A Visual Optimization Specification Language", in *Proceedings of the International Workshop AVI '92*, 1992, pg 289.

In order to fully exploit the parallelism of parallel processors, code transformations must be performed. These transformations have traditionally been shown in text code. We present a visual optimization specification language, VOSpeL, which provides the user with a uniform model to visually specify and perform code transformations. The wide range of applicability of VOSpeL is seen through several examples of the specification of traditional and parallelizing optimizations. VOSpeL can be used as part of a visualization system for transformed parallelized programs. A multi-level browser, which has been implemented, can be used to browse any block of statements in one program view and the corresponding code of another program view will be highlighted. VOSpeL is a graphical form of a General Optimization specification Language (GOSpeL) and is based primarily on the Program Dependence Graph (PDG) model GOSpeL, used to implement an automatic optimizer generator, permits the uniform specification both traditional and parallelizing optimizations by using common constructs. The VOSpeL specification takes advantage of the PDG representation to provide the user with a visual model that is clear, allows different level program representations and enables the user to actually see what statements can be executed in parallel.

[Duisberg87] Robert Adamy Duisberg, "Visual Programming of Program Visualizations", in *1987 Workshop on Visual Languages*, 1987, pgs 55–66.

Algorithm animation promises to be useful in software engineering environments. But fulfillment of the promise will require facilities for the easy construction of animations, for example to be created on the fly in the course of debugging. This paper describes exploratory work toward a system allowing "animation by demonstration", that is capturing the user's gestures involved in direct mouse manipulation of a picture in the environment of a drafting program. Such gestures may be related to specific events in the text of a program, again by gestural indication. Execution of the program being animated then involves a recompilation of the "instrumental" source code into a version which include the graphics procedure calls expanded in-line. The user is thus spared the writing of any textual graphics code.

[Earnshaw93] R.A. Earnshaw, M.A. Gigante, and H. Jones, **Virtual Reality Systems**, 1993.

[Edel88] M. Edel, , "The Tinkertoy Graphical Programming Environment", in *IEEE Transactions on Software Engineering*, 1988, pgs 1110 – 1115.

Tinkertoy is a graphic interface to Lisp, where programs are "built" rather than written, out of icons and flexible interconnections. It is exciting because it represents a computer/user interface that can easily exceed the interaction speed of the best text-based language editors and command languages. It also provides a consistent framework for interaction across both editing and command execution. Moreover, because programs are represented graphically, structures that do not naturally conform to the text medium can be clearly described, and new kinds of information can be incorporated into programs and program elements.

[ElKassas92] S. El-Kassas, "HIRG: A Model for Defining Hierarchical Visual Languages", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 237.

This paper addresses the issue of the formal definition of hierarchical visual languages. It is motivated by interest in formal system development and the use of visual languages as formal specification tools. The paper presents a formal model of hierarchical visual languages. This model is then extended to enable the integration of textual and visual notations.

[Ellis90] Margaret A. Ellis and Bjarne Stroustrup, **The Annotated C++ Reference Manual. (ANSI Base Document)**, 1990.

This is a complete reference for the C++ language. In addition to the manual, approved as the base document of ANSI standardization for the language, are annotations and commentary. These discuss what is not included, why certain features are defined as they are, and how one might implement particular features. Comparisons with C and examples explain the more subtle points of the language.

[Ellis93] John R. Ellis and David L. Detlefs, *Safe, Efficient Garbage Collection for C++*, 1993.

We propose adding safe, efficient garbage collection to C++, eliminating the possibility of storage-management bugs and making the design of complex, object-oriented systems much easier. This can be accomplished with almost no change to the language itself and only small changes to existing implementations, while retaining compatibility with existing class libraries.

Our proposal is the first to take a holistic, system-level approach, integrating four technologies. The language interface specifies how programmers access garbage collection through the language. An optional safe subset of the language automatically enforces the safe-use rules of garbage collections and precludes storage bugs. A variety of collection algorithms are compatible with the language interface, but some are easier to implement and more compatible with existing C++ and C implementations. Finally, code generator safety ensures that compilers generate correct code for use with collectors.

[Finzer84] W. Finzer and L. Gould, "Programming by Rehearsal", in *BYTE magazine*, 1984, pgs 187 – 210.

Rehearsal is a visual programming environment that nonprogrammers can use to create educational software. The emphasis in this graphical environment is on programming visually; only things that can be seen can be manipulated. The design and programming process consists of moving "performers" around on "stages" and teaching them how to interact by sending "cues" to one another. (etc.)

[Fischer93] Alice E. Fischer and Frances S. Grodzinsky, **The Anatomy of Programming Languages**, 1993.

This is a comprehensive text which attempts to dissect language and explain how a language is really built. The first eleven chapters cover the core material: language specification, objects, expressions, control, and types. The more concrete aspects of each topic are presented first, followed by a discussion of implementation strategies and the related semantic issues. Later chapters cover current topics, including modules, object-oriented programming, functional languages, and concurrency constructs.

[Ford92] Lindsey Ford, Brian Lings, and Wendy Milne, *Addressing the Software Engineering Assessment Crisis*, XX.

There is a need to be able to assess whether students have acquired real software engineering knowledge and skills. It is all very well pointing to a student succeeding in individual courses on topics such as systems analysis, HCI, and formal systems, but can the student use this disparate knowledge coherently on a large project? Furthermore a student may be able to work efficiently on an individual project but can the same student work within a team-based project environment with staged deadlines to meet and perhaps difficult fellow team members? We report on our attempt to impart and assess these skills and knowledge. Early feedback suggests that some real learning has taken place.

[Ford93a] Lindsey Ford, *Automatic Software Visualization Using Visual Arts Technics, Research Report 279*, 1993.

The problems of reconciling spatial and temporal dimensions of source code have been with us since the advent of programming. Although the notion of visually representing the internal states and actions of a computer is not new but for algorithm animation, at least, it involves a heavy time penalty on designing and implementing the animation. We have attempted to automate this process by providing a general mechanism for parsing and transforming source code in order to yield standard representations. General visualization mechanisms are then applied to these representations. The problem of space and time persist, however, and we have used techniques from the visual arts, particularly theatre and film, to address them. Our results encourage us to believe that techniques such as these are relevant for completely automating the visualization of very large programs.

[Ford93b] Lindsey Ford, *Separation of Concerns in Teaching Interactive Systems Designs*, 1993.

Position Statement

There are two tricks to teaching design of interactive systems:

• separate two concerns (Frontend and BackEnd) at the right time,

• do not separate software engineering and HCI concerns.

Paper goes to explain and describe.

[Ford93c] Lindsey Ford, *How Programmers Visualize Programs*, 1993. How does a programmer see a computer language? What does a program look like? How would a programmer express these visualizations given a set of graphic and animation creation tools? We explore these questions with learners of object-oriented programming. The learners were provided with the tools and wrote programs to animate features of the language C++. We present the results and conclude that: (1) learners use various abstraction when visualization; (2) a study of programmers' visualizations provides a complementary view to textual-based empirical studies of programmers; (3) programmers frequently represent the same textual programming construct in different visual forms; (4) visualization provides a framework for studying learners' misconceptions; and (5) visualization exercises for learners appear to foster programming skills.

[Ford93d] Lindsey Ford, *Goofy Animation System*, 1993.

The ideas of John T. Stasko and his software system, POLKA, have been instrumental in the development of the Goofy system. POLKA is a system that provides smooth 2 1/2 dimensional colour animations on top of the X11 Window System, and similar 3-dimensional effects on workstations. POLKA provides functions that can be called directly by the animator's code to produce animations.

Goofy provides two interfaces to POLKA, one via a file, the other a functional interface. These interfaces allow animation providers to access POLKA indirectly through a higher level interface. Also, during the development of Goofy, we are adding some extra functinoality to POLK software—for example, sound, various graphic transformations, and moving a "camera". POLKA and Goofy together provide a comprehensive environment for very sophiisticated animations.

The Goofy language provides the animation designer with object definition constructs, object movement descriptors, an object attribute change facility, a method for choreographing events, and several windows for animation.

Goofy uses the "theatre" metaphor to facilitate its use by the animation designer: the theatre has a number of stages (windows), each stage has a background (colour), a play (animation) is enacted through a number of scenes, ascene may involve action on any of the stages by a cast of characters (objects). Each scene is scripted with timed events (actions), and actors (objects) perform the events.

[Ford93e] Lindsey Ford, *Interactive Learning and Researching with Visualization*, 1993.

Can the use of visualization promote links between learners, teachers, and researchers? Could such links improve learning, the quality of teaching, or stimulate new research? We report on our tentative experiment in this direction using animation software and conclude that it does indeed promote links and stimulate research. We present our plans for the future.

[Ford93f] Lindsey Ford and Daniel Tallis, "Interacting Visual Abstractions of Programs", in *1993 IEEE Symposium on Visual Languages*, 93, pgs 93–97.

What visual program abstractions support programming? We explore this question for object-oriented programming with reference to preogramming tasks such as modification engineering and program development. We present ten related abstractions (views) of a program using visual constructs based on empirical and observational studies. We explain the dynamic and interactive nature of the views and suggest how they would be used in programming tasks.

[Ford93g] Lindsey Ford and Daniel Tallis, "Interacting Visual Abstractions of Programs", in *1993 IEEE Symposium on Visual Languages*, 1993.

This is the published version of the paper referenced [Ford93f]

What visual program abstractions support programming? We explore this question for object-oriented programming with reference to preogramming tasks such as modification engineering and program development. We present ten related abstractions (views) of a program using visual constructs based on empirical and observational studies. We explain the dynamic and interactive nature of the views and suggest how they would be used in programming tasks.

[Frakes90] Bill Frakes, *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, 1990.

Table of contents:

Preface, Library Methods I, Reuse Environments, Design & Adaptation I, Library Methods II, Library Methods III, Validation, Compilable Specs, Domain Specific Language, Domain Analysis I, Domain Analysis II, Design and Adaptation II, Attendees.

[Furnas91] George W. Furnas, "New Graphical Reasoning Models for Understanding Graphical Interfaces", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 71.

This paper aspires to make three points: (1) that certain graphical interfaces are especially easy to learn and use, (2) that special graphical deduction/computation systems are possible, and (3) that perhaps points (1) and (2) are intimately related, i.e., that graphical interfaces may be especially useful because they engage special human graphical reasoning processes.

[GACote94] R. GA Cote, *Desktop Telephony*, 1994, pgs 151 – 152.

Describes PhonePro, a Macintosh program for creating automated, programmable voice-mail systems. Offers a graphical scripting language. PhonePro works with any analog telephone system. Offered commercially by Cypress Research.

[Glassman93] Steven C. Glassman, "A TURBO ENVIRONMENT FOR PRODUCING ALGORITHM ANIMATIONS", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 32–36.

We describe GEF, a fast turnaround environment for developing algorithms animations. GEF is an interpreted programming language with operations for defining graphic objects and animations.

[Glinert86] Ephraim P. Glinert , "Towards "Second Generation" Interative, Graphical Programming Environments", in *IEEE 1986 Workshop on VIsual Languages*, 1986, pgs 61–69.

The study and design of interactive, graphical programming environments (IPGEs) capable of providing a high bandwidth for human-computer communication is still in its infancy. For the vast majority of professional programmers, the supremacy of conventional, textual programming remains essentially unchallenged at present. We foresee that, in the near future, the seemingly divergent avenues along which research related to programming environments has progressed in recent years will merge, and thus lead to a "new generation" of IGPEs which will prove beneficial to a greater proportion of the community of computer users including, in particular, both novices and those who may be termed "computing specialists". BLOX, our mixed textual-graphical programming methodology currently under development, is proposed as a possible way to resolve certain issues relating to these new environments which many presently see as problematical.

[Glinert89] E. P. Glinert , and D.W. McIntyre, "The User\xd5 s View of SunPict, an Extensible Visual Environment for Intermediate-Scale Procedural Programming", in *Proceedings. Fourth Israel Conference on Computer Systems and Software Engineering*, 1989, pgs 49 – 58.

[Glinert90a] Ephraim P. Glinert , **Visual Programming Environments, Applications and Issues**, 1990.

Like its companion text, Visual Programming Environments, Paradigms and Systems, this collection of papers is one of the most authoritative collections in the field.

[Glinert90b] Ephraim P. Glinert , **Visual Programming Environments, Paradigms and Systems**, 1990.

Like its companion text, Visual Programming Environments, Applications and Issues, this collection of papers is one of the most authoritative collections in the field. The editor is internationally recognized for his expertise and insight in visual programming.

[Gloor92] Peter A. Gloor , "AACE—Algorithm Animation for Computer Science Education", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 25.

This paper describes AACE, a methodology for education algorithm animation that we developed while building an integrated hypermedia algorithm animation environment extending a fundamental algorithms textbook. After presenting the general structure of AACE, we discuss questions of the ideal user interface for algorithm animations. We compare our approach for the development of algorithm animations that we call "structure-based" with the more conventional "unified view" -based approach.

[Goldberg89] Adele Goldberg and David Robson, **SMALLTALK-80 The language.**, 1989.

The first part of this book introduces the Smalltalk-80 approach to information representation and manipulation; it also provides an overview of the syntax of the language. The second section contains specifications of the kinds of objects already present in the Smalltalk-80 programming environment. New kinds of objects can be added by a programmer, but a wide variety objects come with the standard system. An example of adding new kinds of objects to the system is included in the third part; this example describes the addition of an application to model discrete, event-driven simulations such as car washes, banks, or information systems.

[Golin93] Eric J. Golin and Tom Magliery, "A Compiler Generator for Visual Languages", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 314–321.

Building a compiler for a visual programming language is a significant task, and is complicated by the difficulty in handling visual syntax. Object-oriented picture layout grammars are a grammar formalism for defining visual syntax that users C++ to define graphical attributes and constraints. SPARGEN is a compiler-compiler that automatically generates a visual langauge compiler from an OOPLG specification. This paper describes how SPARGEN can be used to construct compilers for visual programming languages.

[Green95] Thomas Green , "Noddy's Guide to Visual Programming", in *The British Computer Society — Human-Computer Interaction Group "Interfaces", Autumn '95*, 1995.

Discusses what visual programming is and is not. Offers a coherent overview on the topics, issues, style concerns, with extensive references.

[Grover91] Mark D. Grover, Ph.D., "The Pleasure and Pain of Persistence", in *Addendum to the Proceedings OOPSLA 91*, 1991, pgs 107 – 110.

SynchroWorks(TM) by Oberon Software; "Oberson Software is constructing a product which creates and manages complex databases of persistent objects. This presentation describes useful experiences in the design and development of this product.... The product provides a convenient software environment for configuring and personalizing shared data and applications among information management professionals. "

[Grundy91] John C. Grundy and John G. Hosking, *Integrated Object-Oriented Software Development in SPE*, XX.

SPE is a software development environment which supports multiple textual and graphical views of a program. Views are kept consistent with one another using a mechanism of update records. SPE is useful throughout all phases of the software development life-cycle. It provides support for conceptual level object-oriented analysis and design using diagrams, visual and textual programming, hypertext-based browsing, and visual debugging, together with a modification history. SPE is implemented as a specialization of an object-oriented framework and provides an environment for Snart, an object-oriented programming language.

[Grundy93a] John Collis Grundy, **Multiple Textual and Graphical Views for Interactive Software Development Environments**, 1993.

Diagram construction can be used to visually analyse and design a complex software system using natural, graphical representations describing high-level structure and semantics. Textual programming can specify detailed documentation and functionality not well expressed at a visual level. Integrating multiple textual and graphical views of software development allows programmers to utilise both representations as appropriate. Consistency management between these views must be automatically maintained by the development environment.

MViews, a model for such software development environments, has been developed. MViews supports integrated textual and graphical views of software development with consistency management. MViews provides flexible program and view representation using a novel object dependency graph approach. Multiple views of a program may contain common information and are stored as graphs with textual or graphical renderings and editing. Change propagation between program components and views is supported using a novel update record mechanism. Different editing tools are integrated as views of a common program repository and new program representations and editors can be integrated without affecting existing views.

A specification language for program and view state and manipulation semantics, and a visual specification language for view appearance and editing semantics, have been developed. An object-oriented architecture based on MViews abstractions allows environment specifications to be translated into a design for implementing environments. Environment designs are implemented by specialising a framework of object-oriented language classes based on the MViews architecture. A new language is described which provides object-oriented extensions to Prolog. An integrated software development environment for this language is discussed and the specification, design and implementation of this environment using MViews are described. MViews has also been reused to produce a graphical entity-relationship/textual relational database scheme modeller, a dialogue painter with a graphical editing view and textual constraints view, and various program visualisation systems.

[Grundy93b] J.C. Grundy and J.G. Hosking, "MViews: A Framework for Developing Visual Programming Environments", in *Technology of Object-Oriented Languages and Sytems (TOOLS 12)*, 1993.

[Harmon91] Paul Harmon, and Brian Sawyer, **ObjectCraft. A Graphical Programming Tool for Object-Oriented Applications,** 1991.

This book is the manual for ObjectCraft, software providing "a tool for doing object-oriented programming visually", automatically converting code to C++ or Turbo Pascal.

[Helm91] Richard Helm, Kim Marriott, and Martin Odersky, "Building Visual Language Parsers", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 105.

Notepad computers promise a new input paradigm where users communicate with ocmputers in visual languages composed of handwritten text and diagrams. A key problem to e solved before such an interface can be realized is the efficient and accurate recognition (or parsing) of handwritten input. We present techniques for building visual language parsers based on a new formalism, constrained set grammars. Constrained set grammars provide a high-level an d declarative specification of visual languages and support the automatic generation of efficient parsers. These techniques have been used to build parsers for several representative visual languages.

[Heydon90] A. Heydon, M. Maimone, J. Tygar, J. Wing, and A. Zaremski, "Miro: Visual Specification of Security", in *IEEE Transactions on Software Engineering*, 1990, pgs 1185 – 1197.

Miro is a set of language and tools that support visual specification of file system security. We present two visual languages; the instance language, which allows specification of file system access, and the constraint language, which allows specification of security policies. We also describe tools we have implemented and give examples of how our languages can be applied to real security specification problems.

[Heydon94] Allan Heydon and Greg Nelson, *The Juno-2 Constraint- Based Drawing Editor*, 1994.

Constraints are an important enabling technology for interactive graphics applications. However, today's constraint-based systems are plagued by several limitations, and constraints have yet to live up to their potential.

Juno-2 is a constraint-based double-view drawing editor that addresses some of these limitations. Constraints in Juno-2 are declarative, and they can include non-linear functions and ordered pairs. Moreover, the Juno-2 solver is not limited to acyclic constraint systems. Juno-2 also includes a powerful extension language that allows users to define new constraints. The system demonstrates that fast constraint solving is possible with a highly extensible, fully declarative constraint language.

The report describe what it is like to use Juno-2, outlines the methods that juno-2 uses to solve constraints, and discusses its performance.

[Hilton90] Michael Lee Hilton, *Implementation of Declarative Languages*, 1990.

This dissertation is concerned with the design of computing architectures which support the implementation of declarative programming languages. Declarative languages, which include the functional and logic programming languages, are based on mathematical theories of computation. These mathematical foundations endow declarative languages with elegant semantics and expressiveness, but make efficient implementation on conventional von-Neumann computers difficult.

[Hirakawa87] M. Hirakawa, S. Iwata, I. Yoshimoto, M. Tanaka, and T. Ichikawa, "HI-VISUAL Iconic Programming", in *1987 Workshop on Visual Languages*, 1987.

We earlier proposed a visual programming language, HI-VISUAL, which was designed to attain interactive iconic programming. Programming in HI-VISUAL is carried out simply by arranging icons on the display screen.

In this paper, we extend HI-VISUAL as an environment for iconic programming by providing the following facilities: (1) navigation for program development and system operations, (2) interpretation mechanisms for icon programs and system operations, based on the object-oriented concept, (3) design of user-defined interfaces, (4) top-down development of programs, and (5) integration of existing (sub)systems.

The architecture of HI-VISUAL for programming, execution, and management of icon programs will also be presented.

[Hirakawa88] M. Hirakawa, "A Framework for Construction of Icon Systems", in *1988 IEEE Workshop on Visual Languages*, 1988, pgs 70–77.

Iconic programming is effective for attaining higher man-machine interaction from the viewpoints of both universality and efficiency. In this paper, the authors propose a framework for the construction of icon systems.

An icon system is composed of icons and rules. Icons represent real objects such as sales books, folders, calculators, etc. Functions associated with an object are specified in the icon representing the object. Icons therefore have both data and function properties. Icons in fact may have several functions. The behavior of an icon is not fixed but is determined at the time of programming by being combined with another icon.

Rules are provided to make flexible interpretation of icons feasible depending on the application, the status of the system, and so on. The behavior of the system can be changed by replac ing icons and/or rules with new ones.

Implementational issues of the system are also described. A system prototype is now in an actual operation on a workstation in our laboratory environment.

[Hirakawa90a] Masahito Hirakawa, Makoto Yoshimi, and Tadao Ichikawa, "A Universal Language System for Visual Programming", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 156–161.

Development of visual programming systems is usually carried out by means of a conventional text-based programming language such as C or Lisp. This imposes a large burden on the system designer because of the difference in the two language models. To lighten the burden, an environment which supports construction of visual programming systems is desirable.

In this paper, the authors present a universal language system for visual programming. Any specific visual programming system can easily be constructed by using the system. The system is based on our previous studies of VPS/VPSM, and extended by providing (1) a unified, object-oriented model of visual elements management and (2) support tools which enable the system designer to construct visual programming systems more easily.

[Hirakawa90b] M. Hirakawa, M. Tanaka, and T. Ichikawa, "An Iconic Programming System, HI-VISUAL", in *IEEE Transactions on Software Engineering*, 1990, pgs 1178 – 1184.

The use of icons is effective for attaining higher man-machine interaction in programming, from the viewpoints of both universality and efficiency. In this paper, the authors propose a new framework for icon management and an iconic programming based on it.

In the framework, icons represent real objects or the concepts already established in a target application environment, whereas icons representing functions are not provided. A function is represented by a combination of two different icons. Each icon can take an active or a passive role against the other. The role sharing is determined dynamically depending on the environment in which the icons are activated.

The framework for icon management mentioned above, which is quite object-oriented, if first proposed, and then an iconic programming system named HI-VISUAL is presented on the basis of the framework. Programming in HI-VISUAL and implementational issues of the system prototype, now in actual operation in our laboratory environment, are extensively discussed.

[Holt90] C.M. Holt, "Viz: A Visual Language Based on Functions", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 221–226.

A significant feature of visual languages is that one may use geometric position to represent non-linear structure directly, in contract to "algebraic" languages that are restricted to totally ordered sequences of symbols. Directed graphs are well-suited to the description of partial orders of applications and events, as found in functional and concurrent models; the use of visual techniques therefore has the potential to eliminate a level of encoding on the part of the programmer. This paper introduces some conventions that are intended to allow a straightforward representation of a functional semantics, while being general enough to be extendable to more general semantic models.

[Holzle93] Urs Holzle , *Integrating Independently-Developed Components in Object-Oriented Languages*, 1993.

Object-oriented programming promises to increase programmer productivity through better reuse of existing code. However, reuse is not yet pervasive in today's object-oriented programs. Why is this so? We argue that one reason is that current programming languages and environments assume that components are perfectly coordinated. Yet in a world where programs are mostly composed out of reusable components, these components are not likely to be completely integrated because the sheer number of components would make global coordination impractical. Given that seemingly minor inconsistencies between between individually designed components would exist, we examine how they can lead to integration problems with current programming language mechanisms. We discuss several reuse mechanisms that can adapt a component in place without requiring access to the component's source code and without need to re-typecheck it.

[Horowitz95] Ellis Horowitz , **Programming Languages: A Grand Tour. 2nd edition**, 1995, pg 1985.

This comprehensive anthology presents a twenty-year overview of programming languages and contains an organized collection of articles and language reference materials for students of programming languages and professional computer scientists. Beginning with a history of programming languages, the book chronicles the appearance of each new language and its contributions, and emphasizes the difficulty of successful language design. It includes significant papers on the ALGOL family of languages, applicative programming languages, data abstraction languages, Ada (the new language developed by the US Department of Defense), and languages with concurrency features. This new edition of Programming Languages: A Grand Tour now includes the new February 1983 edition of the Reference Manual for the Ada Programming Language, as well as complete language reference manuals for ALGOL60, ALGOL-3, Lisp 1.5, Modula, and C.

[Horton94] William Horton , **The Icon Book. Visual Symbols for Computer Systems and Documentation**, 1994.

User-interface icons are much more than on-screen decorations. They plan an integral role in enhancing end-user productivity and an application's overall success. Written by a seasoned design professional, the book presents clear, step-by-step guidelines for designing instantly recognizable, fully understandable, and reliably memorable computer icons and icon sets for domestic and international use. The book is heavily illustrated, with research-based accounts on every aspect of the icon design process.

[HP88] Hewlet-Packard, Corvallis, *SysRPL Programming Guide*, 1988.

Everything you ever wanted to know about the underlying interface of the HP 48 series calculators.

[Hsia88] Yenm-Teh Hsia, "Construction and Manipulation of Dynamic Icons", in *1988 IEEE Workshop on Visual Languages*, 1988, pgs 78–83.

A dynamic icon is an icon whose graphical representation is computed and varies over time with changes in the properties of the object it represents. Dynamic icons make possible use of iconic representation of objects whose dynamic behavior must be reflected in their representation. The visual langauge PT incorporates dynamic icons to model dynamic object behavior in executing programs. This use of dynamic icons provides a natural program animation mechanism. This paper discusses dynamic icons and their use in PT.

[Hudson] Scott E. Hudson and John T. Stasko, *Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions.*

Animation can be a very effective mechanism to convey information in visualization and user interface settings. However, integrating animated presentations into user interfaces has typically been a difficult task since, to date, there has been little or no explicit support for animation in window systems or user interface toolkits. This paper describes how the Artkit user interface toolkit has been extended with new animation support abstractions designed to overcome this problem. These abstractions provide a powerful but convenient base for building a range of animations, supporting techniques such as simple motion-blur, 'squash and stretch', use of arcing trajectories, anticipation and follow through, and 'slow-in/ slow-out' transitions. Because these abstractions are provided by the toolkit they are reusable and may be freely mixed with more conventional user interface techniques. In addition, the Artkit implementation of these abstractions is robust in the face of systems (such as the X Window system and Unix) which can be ill-behaved with respect to timing considerations.

[Humenn90b] Paul R. Humenn, *User's Manual for the D Programming Language, Version 3.2*, 1990.

This is definitely a user's manual.

[Hyrskykari87] Aulikki Hyrskykari, and Kari-jouko Raiha, "Animation of Algorithms Without Programming*", in *1987 Workshop on Visual Languages*, 1987.

Animation of data has been found to be a useful tool for teaching and for developing new algorithms. However, the use of animations has been limited because of the large amount of programming needed. We are implementing a system that generates the animation automatically on the basis of a simple, declarative specification. In this report we present a brief survey of current work on animation of algorithms and introduce the specification method used in our system.

[Ichikawa84] Tadao Ichikawa, and Shi-Kuo Chang, *IEEE 1984 Workshop on Visual Languages*, 1984.

EEE 1984 Workshop on Visual Languages, Dec. 6–8, 1984, Hiroshima, Japan

[Ichikawa86] Tadao Ichikawa, *IEEE 1986 Workshop on VIsual Languages*, 86.

IEEE 1986 Workshop on VIsual Languages, June 25-27, 1986, Dallas, TX, USA

[Ichikawa92] T. Ichikawa, and H. Tsubotani, **Language Architectures and Programming Environments**, 1992.

This is a collection of papers which the editors consider milestones in relation to past research; the newest work isn't necessarily included, but it's the quality that counts. The editors feel they offer a solid historical overview of the growth of the technology and the philosophical aspects involved. The collection offers interesting and unusual insight into the approach past pioneers took in solving problems we still work with today.

[Ingalls88] D. Ingalls, S. Wallace, Yu-Ying Chow, F. Ludolph and K. Doyle, "Fabrik A Visual Programming Environment", in *ACM SIGPLAN 3rd Annual Conference on Object-Orientated Programming Systems, Languages, and Applications (OOPSLA 88)*, 1988, pgs 176 – 190.

Fabrik is a visual programming environment—a kit of computational and user-interface components that can be "wired" together to build new components and useful applications. Fabrik diagrams utilize bidirectional dataflow connections as a shorthand for multiple paths of flow. Built on object-oriented foundations, Fabrik components can compute arbitrary objects as outputs. Music and animation can be programmed in this way and the user interface can even be extended by generating graphical structures that depend on other data. An interactive type system guards against meaningless connections. As with simple dataflow, each Fabrik component can be compiled into an object with access methods corresponding to each of the possible paths of data propagation.

[Inmon89] W.H. Inmon , **Data Architecture: The Information Paradigm**, 1989.

In this book the author lays the foundation of an information systems architecture, defines the information paradigm and data architecture and then discusses the evolution of the information paradigm.

[Jamsek90] Damir A. Jamsek, *The WINTER Architecture: Support for a Purely Declarative Programming Language*, 1990.

Combining functional and logic programming paradigms in a single environment leads to a powerful and expressive programming language. The WINTER abstract machine provides an execution environment that efficiently realizes the goals of such a programming system. The WINTER abstract machine draws much of its inspiration from currently popular abstract machines being developed for either functional or logic programming. It is novel in its unification of architectures that initially seem to be addressing entirely different programming paradigms. This unification leads to a common and consistent treatment of both functional and logic programming.

The main technical results presented in this work include several aspects in the development of the WINTER architecture. The primary result is the definition of the WINTER architecture including the structure and operational semantics of the machine designed. In addition, the semantics of the programming language are presented formally as an axiomatic term rewriting system, thus providing a basis for determining the correctness of the WINTER design.

The WINTER machine is presented in a way which lends itself to formal verification. That is, the syntax and semantics of the language being supported are formally presented, as well as the structure and operational semantics of the WINTER machine. This is expressed in a form suitable for mechanical verification by any one of a number of currently available automated proof assistants. This formal presentation provides a sound engineering basis from which to address further topics of machine design to be encountered in the subsequent development of refined versions of the WINTER machine.

[Jeffries91] Robin Jeffries, James R. Miller, Cathleen Wharton, and Kathy Uyeda, "User Interface Evaluation in the Real World: A Comparison of Four Techniques", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 119.

A user interface (UI) for a software product was evaluated prior to its release by four groups, each applying a different technique: heuristic evaluation, software guidelines, cognitive walkthroughs, and usability testing. Heuristic evaluation by several UI specialists foudn the most serious problems with the least amount of effort, although they also reported a large number of low-priority problems. The relative advantages of all the techniques are discussed, and suggestions for improvements in the techniques are offered.

[Jerding94] Dean F. Jerding, and John T. Stasko, *Using Visualization to Foster Object-Oriented Program Understanding*, 1994, pgs 1–15.

Software development and maintenance tasks rely on and can benefit from an increased level of program understanding. Object-oriented programming languages provide features which facilitate software maintenance, yet the same features often make object-oriented programs more difficult to understand. We support the use of program visualization techniques to foster object-oriented program comprehension. This paper identifies ways that visualization can increase program understanding, and presents a means for characterizing both static and dynamic aspects of an object-oriented program. We then describe the implementation of a prototypical tool for visualizing the execution of C++ programs. Based on this work, we define a framework for the visualization of object-oriented sw which requires little or no programmer intervention and provides a mechanism which allows users to focus quickly on particular aspects of the program.

[Jungert88] E. Jungert, *1988 IEEE Workshop on Visual Languages*, 1988.

1988 IEEE Workshop on Visual Languages, October 10-12, 1988, Pittsburgh, PA, USA

[Jungert93] Erland Jungert, "Graqula—A Visual Information-flow Query Language for a Geographical Information System", in *Journal of Visual Languages and Computing*, 1993, pgs 383–401.

The query language Graquala, described and discussed here, is designed for a geographical information system. The basic map data structure to which Graquala has been adapted is a homogeneous raster-based structure of run-length-code (RLC) type. Because of the homogeneity of the map data structure, queries can be applied in a more or less generic way; that is, queries can be applied to all map data regardless of their type. For this reason it has been possible to design a visual data-flow query language that allows compound queries which can become quite complex. In this work, Graqula and some elements of its environment, including a brief overview of the basic map data structure, will be discussed.

[Kado92] M. Kado, M. Hirakawa, and T. Ichikawa, "HI-VISUAL for Hierarchical Development of Large Programs", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 48.

In recent years, the demand for the development of programming environments friendly non-experts has increased. At Hiroshima University, we have been developing an iconic programming environment called, HI-VISUAL, where program construction is carried out by arranging icons on the screen. In this paper, an extension of HI-VISUAL programming facilities for enabling users to develop large programs is investigated. We first present a model for supporting hierarchical development of large programs. Program development is achieved in the model by means of two primitive components, agent and frame. An agent is a special icon which can manage user-defined programs, while a frame is a container for managing a set of icons (including agents) allowing the user to construct programs hierarchically. Implementational issues of a system prototype are also discussed. In the system, an office metaphor is applied for realization of the hierarchical program development model, where an agent and a frame correspond to an employee and an office room, respectively. The user in the real world operates icons/agents in a virtual office room and specifies a sequence of tasks, i.e. a program. Here, multiple users can have individual office rooms (frames) which are connected to each other through a computer network. This means that members in a project are allowed to work cooperatively for development of large programs.

[Katiyar94] Dinish Katiyar, David Luckham, and John Mitchell, "A type system for prototyping languages", in *Conf. Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994, pgs 138–150.

RAPIDE is a programming language framework designed for the development of large, concurrent, real-time systems by prototyping. The framework consists of a type language and default executable, specification and architecture languages, along with associated programming tools. We describe the main feature of the type language, its intended use in a prototyping environment, and rationale for selected design decisions.

[Keene88] Sonya E. Keene, **Object-Oriented Programming in COMMON LISP, A Programmer's Guide to CLOS**, 1988.
In the words of Daniel G. Bobrow, from Xerox Palo Alto Research Center, "This book is a superb introduction to programming with the Common Lisp Object System. It presents clearly, through realistic programming examples, the important material in CLOS. It also indicates how readers should think about constructing such programs themselves, and provides style hints and discusses tradeoffs in the use of different features. Its discussion of how to design and document an object-oriented program will also be of interest of programmers who work in other languages."

[Kent92] William Kent, "User Object Models", in *OOPS Messenger*, 1992, pgs 10–25.
Object users and developers have different needs, contributing to the plethora of object models. This paper focuses on user object models, differentiating them from developer models, and outlines a spectrum of characteristics which can provide a basis for comparing and reconciling different user models.

[Khoral93] Khoral Research, Inc., *Cantata The Khoral Visual Programming Environment*, 1993, pgs 1–87.
Visual Programming Manual. Use of the Khoros system.

[Khoros95] Khoral Research, Inc., *Installing Khoros, an Installation Guide. Appendix A: Cantata Operators*, 1995.
Installation Guide

[Kimura92] Takayuki Dan Kimura, "Hyperflow: A Visual Programming Language for Pen Computers", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 125.

The design philosophy of th Hyperflow visual programming language and an overview of its semantic model are presented. The concept of visually interactive process, vip, is introduced as the fundamental element of the semantics. Vips communicate with each other through exchange of signals, either discrete or continuous. Each vip communicates with the user through its own interface box by displaying on the box information about the vip and by receiving information pen-scribed on the box. There are four different communication modes: mailing, posting, channeling, and broadcasting. Mailing and posting are for discrete signals and channeling and broadcasting are for continuous signals. Simple Hyperflow programs are given including a specification for the Line-Clock device driver.

[Klausner91] Sanford B. Klausner, "The Cubicon Project", in *Addendum to the Proceedings OOPSLA 91*, 1991, pgs 105 –107.

[Koike93] Hideki Koike and Hirotaka Yoshihara, "Fractal Approaches for Visualization Huge Hierarchies", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 55–60.

This paper describes fractal approaches to the problems which associate with visualizing huge hierarchies. The geometrical characteristic of a fractal, self-similarity, allows users to visually interact with a huge tree in the same manner at every level of the tree. The fractal dimension, a measure of complexity, makes it possible to control the total amount of displayed nodes. A prototype visualization system for UNIX directoris is also shown.

[Kopache88] Mark E. Kopache and Ephraim P. Glinert, "C^2: A Mixed Textual/ Graphical Environment for C", in *1988 IEEE Workshop on Visual Languages*, 1988, pgs 231–238.

A visual programming environment for a subset of the C language is described. The C^2 environment, as it is called, runs on a personal workstation with high resolution graphics display. Both conventional textual code entry and editing, and program composition by means; of an experimental hybrid textual/graphical method, are supported and coexist side by side on the screen at all times. The built-in text editor incorporates selected UNIX vi commands in conjunction with a C syntax interpreter. Hybrid textual/graphical program composition is facilitated by a BLOX-type environment in which graphical icons represent program structures and text in the icons represents user-supplied parameters attached to those structures. The two representations are coupled, so that modification entered using either one automatically generate the appropriate update in the other. Although not all of the C language is yet supported, C^2 is not a toy system. Textual files that contain C programs serve as input and output. Graphical representations serve merely as interally-generated aids to the programmer, and are not stored between runs.

[Korfhage84] Robert R. Korfhage and Margaret A. Korfhage, "The Nature of Visual Languages", in *IEEE 1984 Workshop on Visual Languages*, 1984, pgs 177–183.

Most natural and programming languages available today have a strongly linear structure. We consider the implications that htis has for visual representation and manipulation, and suggest some characteristics of a class of languages better suited to these tasks.

[Korfhage87] Robert R. Korfhage, *1987 Workshop on Visual Languages*, 1987.

1987 Workshop on Visual Languages, August 19–21, Linköping, Sweden

[Kutty] G. Kutty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna, *A Graphical Methodology for Concurrent System Design*, 93 or later.

Graphical Interval Logic is a visual formalism used to specify and reason about concurrency in software and hardware designs. In this paper we illustrate the use of Graphical Interval Logic in the design of a concurrent system, a simple communication protocol for reliable sequenced communication over an unreliable medium. We also describe a graphical toolkit that we have developed for Graphical Interval Logic.

[Lamport94] Leslie Lamport, *Processes are in the Eye of the Beholder*, 1994.

A two-process algorithm is shown to be equivalent to an N-process one, illustrating the insubstantiality of processes. A completely formal equivalence proof in TLA (the Temporal Logic of Actions) is sketched.

[Larkin90] Timothy S. Larkin and Raymond I. Carruthers, *HERMES*, 1990.

SERV, a Simulation Environment for Research Biologists, is a tool for constructing models representing continuous processes. SERV has been programmed in Flavors, an OO programmming language imbedded in LISP. Models are themselves objects, consisting of modular components, each representing a process or some aspect of a process. Each component is a black box, with a functional definition which maps its inputs to one or more outputs. Each input is connected to an appropriate output, from which it receives a value. These concepts led to the creation of an early version of SERV in which a text editor was uzed to specify the series of components making up the model, which was text based and linearly organized. Currently, the advantages of Flavors are being further exploited in two main directions.

[Leavens91] Gary T. Leavens, "Introduction to the Literature on Object-Oriented Design, Programming, and Languages", in *OOPS Messenger*, 1991, pgs 40–53.

This paper is an introduction to the literature on object-oriented design, object-oriented programming, a few programming languages (especially C++), and some related topics in language design.

[Levialdi90] Stefano Levialdi , "Congnition, Models & Metaphors", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 69–79.

Man machine interfaces are proviing to be the most delicate and important component for a fruitful and friendly use of complex programs. In order to fully exploit the intellectual power of man new studies in cognition engineering and in the formation of mental models have favoured the use of metaphors for raising the abstraction level of communication and enlarging the community of computer users outside the circle of experts; new studies in metaphor evaluation and utilization in existing commercial programs and in suggested graphical user interfaces are providing new insight in the rapidly growing area of iconic systems and visual langauges which, hopefully, will establish aconstructive methodology for metaphor design.

[Lewis91] J. Bryan Lewis, Lawrence Koved, and Daniel T. Ling, "Dialogue Structures for Virtual Worlds", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 131.

We describe a software architecture for virtual worlds, built on a base of multiple processes communicating through a central event-driven user interface management system. The virtual world's behavior is specified by a dialogue composed of modular subdialogues or rule sets. In order to achieve high flexibility, device remappability and reusability, the rule sets should be written as independent modules, each encapsulating its own state. Each should be designed according to its purpose in a conceptual hierarchy: it can transform a specific device into a generic device, or transform a generic device into an interaction technique, or, at the top level, map interaction techniques to actions.

[Lieberman92] Henry Lieberman, "Dominoes and Storyboard: Beyond "Icons on Strings"", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 65.

Practically since graphic displays were first hooked to computers, the idea of representing computer programs by pictures has attracted researchers. However, to date, most proposals for visual programming languages have adhered to a set pattern: fixed pictures symbolizing program components, connected by lines or arrows symbolizing relationships between the program components. This "icons and strings" approach, while it can be useful, is not the only way of visualizing programs. In this paper, I explore one alternative: representing a program through visual examples of the state of its execution. I present two related techniques: dominoes, which replace the traditional icons as representations of operations; and storyboards, which replace iconic circuitry as the representation of program code. These have been implemented in Mondrian, a graphic editor extensible through programming by example.

[Ludolph88] Frank Ludolph and Yu-Ying Chow , "The Fabrik Programming Environment", in *1988 IEEE Workshop on Visual Languages*, 1988, pgs 222–230.

Fabrik is an experimental interactive graphical programming environment designed to simplify the programming process by integrating the user interface, the programming language and its representation, and the envirnmental languages used to construct and debug programs. The programming language uses a functinoal, bidirectional data-flow model that trivializes syntax and eliminates the need for some traditional programming abstractions. Program synthesis si simplified by the use of aggregate and application-specific operations, modifiable examples, and the direct construction of graphical elements. The user interface includes several features designed to ease the construction and editing of the program graphs. Understanding of both individual functions and program operation are aided by immediate execution and feedback as the program is edited.

[Lundell91] Jay Lundell and Mark Notess, "Human Factors in Software Development: Models, Techniques, and Outcomes", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 145.

We present the results of a survey designed to identify ways that human factors engineers have been successfully involved in software projects. Surveys describing successful and unsuccessful outcomes were returned by 14 human factors engineers and 21 software and documentation engineers at Hewlett Packard. In addition to describing the type of involvement and techniques used, respondents were also asked to define what they considered to be a successful outcome and give their views on what factors contribute to success or failure. The results of this study suggest ways in which the human factors/ R&D partnership can be more effective in current development scenarios.

[Mackay91] Wendy E. Mackay , "Triggers and Barriers to Customizing Software", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 153.

One of the properties of a user interface is that it both guides and constrains the patterns of interaction between the user and the software application. Application software is increasingly designed to be "customizable" by the end user, providing specific mechanisms by which users may specify individual preferences about the software and how they will interact with it over multiple sessions. Users may thus encode and preserve their preferred patterns of use. These customizations, together with choices about which applications to use, make up the unique "software environment" for each individual.

While it is theoretically possible for each user to carefully evaluate and optimize each possible customization option, this study suggests that most people do not. I studied the customization behavior of 51 users of a Unix software environment, over a period of four months. This paper describes the process by which users decide to customize and examines the factors that influence when and how users make those decisions. These findings have implications for both the design of software and the integration of new software into an organization.

[Mackinlay91] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card , "The Perspective Wall: Detail and Context Smoothly Integrated", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 173.

Tasks that involve large information spaces overwhelm workspaces that do not support efficient use of space and time. For example, case studies indicate that information often contains linear components, which can result in 2D layouts with wide, inefficient aspect ratios. This paper describes a technique called Perspective Wall for visualizing linear information by smoothly integrating detailed and contextual views. It uses hardware support for 3D interactive animation to fold wide 3D layouts into intuitive 3D visualizations that have a center panel for detail and two perspective panels for context. The resulting visualization supports efficient use of space and time.

[Mackinlay92] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card , "The Information Visualizer: A 3D User Interface for Information Retrieval", in *Proceedings of the International Workshop AVI '92*, 1992, pg 173.

Advances in computer technology have created new possibilities for information retrieval systems in which user interfaces could play a more central role. Our analysis of the problem suggests that what is needed from the user's point of view is not so much information retrieval itself, but, rather, the amplification of information-based work processes. User interfaces enabled by this technology may be able to amplify work by modifying the cost structure of information used in work. As a consequence, we have attempted to go beyond the usual notion of an information retrieval system to develop an "Information Workspace" that encompasses the cost structure of information from secondary storage to immediate use. As an implementation of the concept, we describe the experimental system, called the Information Visualizer, and its rationale. The system is based on the use of (1) 3D/ Rooms for increasing the capacity of immediate storage available to the user, (2) an animated scheduler based user interface interaction architecture, called the Cognitive Coprocessor, for coupling the user to information agents, and (3) information visualization for interacting with the information structure. The system and its rationale are described.

[Madsen95] Ole Lehrmann Madsen, *Abstract Syntax Structure Editing and Type Browing for Self, draft version 1.2*, 1995.
how to program in Self

[Magnusson94] Boris Magnusson, Bertrand Meyer, Jean-Marc Nerson, and Jean-Francois Perrot, *Technology of Object-Oriented Languages and Systems TOOLS 13*, 1994.
TOOLS (Technology ofObjet-Oriented Languages and Systems) 13. Proceedings of the thirteenth International Conference, Versailles, 1994.

[Mak91] Ronald Mak, **Writing Compilers & Interpreters. An Applied Approach.**, 1991.
This book emphasizes practical skills rather than theory. First it describes writing a set of useful working utility programs, then builds upon these to develop an interpreter, an interactive debugger, and a compiler.

[Manuel95] Darrel Manuel, *The Impossible Dream: Towards General and Automatic Visualizations*, 1995.
Visualization systems at present suffer from a number of significant problem. Many of these are derived from the lack of automatic and general systems which would reduce the amount of work required by the systems' users. In addition the lack of a set of general theories, objects and activities between different systems increases the difficulty in learning how to use them. This paper attempts to describe the levels and forms of automation and generalization inherent in visualization systems, attempting to explain from the problems that exist currently and potentially in the future, with plausible solutions.

[Marcus91] Bob Marcus, *Addendum to the (OOPSLA) Proceedings, AZ '91Birds of a Feather: "END USER"*, 1991, pgs 97–110.
This is additional proceedings from the OOPSLA 91 Conference, October 6–11, 1991, Phoenix, Arizona.

[Marcus92] Bob Marcus, "End Users (Birds of a Feather). OOPSLA '91, Conference on Objected-Oriented Programming systems, Languages, and Applications ", in *Addendum to the Proceedings OOPSLA 91*, 1992.

The main outcome of the end-users BOF at OOPSLA was the formation of an ongoing group tentatively named "Corporate Facilitators of Object-Oriented Technology" (CFOOT). The purpose of the group is to communicate corporate requirements to vendors and consortia. In addition, the group will provide a mechanism for sharing experiences and lessons learned in introducing object technology into large corporations. ...As part of the initial meeting, a list of end-user requirements for vendors was gathered by brainstorming. These are: 1) consider your products as components of larger and/or legacy system. 2) focus on areas where your product can add value. 3) avoid unnecessary proprietary features and interfaces. 4) interoperate with other products and components. 5) plan to interface with multiple legacy platforms, systems, processes and organizations. 6) supply tools for supporting total life cycle project management, methodologies, and metrics. 7) provide robust class libraries to encapsulate non-object-oriented systems. 8) supply tools for reverse engineering and code generation. 9) provide object-oriented screen painters and robust standardized classes for graphical user interfaces. 10) supply source code for class libraries using a reasonable, coherent pricing structure.

[Marks90] Joe Marks , "A Syntax and Semantics for Network Diagrams", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 104–110.

The ability to automatically design graphical displays of data will be important for the next generation of interactive computer systems. The research reported here concerns the automated design of network diagrams, one of hte three main classes of symbolic graphical display (the other two being chart graphs and maps). Previous notions of syntax and semantics for network diagrams are not adequate for automating the design of this kind of graphical display. I present here a new formulation of syntax and semantics for network diagrams that is used in the ANDD (Automated Network Diagram Designer) system. The syntactic formulation differs from previous work in two significant ways: perceptual-organization phenomena are explicitly

[Martin86] George R. R. Martin , **Tuf Voyaging**, 1986.

This science fiction work was enjoyed by the project PI and is the source of name inspirations, such as TUF, ARK, and Cornucopia (from "The Cornucopia of Excellent Goods at Low Prices") for this project.

[Masnavi90] Siamak Masnavi , "Automatic Visualization of the Dynamic Behavior of Programs by Animation of the Language Interpreter", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 17–21.

Understanding the dynamic behavior of programs without good program animation tools can be difficult. Thisis especially true for novie programmers. Unfortunately, most existing tracing tools either only visualize one aspect of execution, provide only textual views of execution, or both. The need for visualization becomes even more urgent when we are dealing with languages which have a complex execution model, e.g., hybrid or multi-paradigm languages. In this paper, we present a system called SeePS for visualization the dynamic behavior of NeWS programs, and describe our experience with design and implementation.

[Masui92] Toshiyuki Masui, "Graphic Object Layout with Interactive Genetic Algorithms", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 74.

Automatic graphic object layout methods have long been studied in many application areas in which graphic objects should be laid out to satisfy the constraints specific to each application. In those areas, carefully designed layout algorithms should be used to satisfy each applications' constraints. However, those algorithms tend to be complicated and not reasonable for other applications. Moreover, it is difficult to add each user's preferences to the layout scheme of the algorithm. To overcome these difficulties, we developed a general-purpose interactive graphic layout system GALAPAGOS based on genetic algorithms. GALAPAGOS is general-purpose because graphic objects are laid out not by specifying how to lay them out, but just by specifying the preferences for the layout. GALAPAGOS can not only lay out complicated graphs automatically, but also allow users to modify the constraints at run time so that user can tell the system their own preferences.

[Matsumura86] Kazuo Matsumura and Shuichi Tayama, "Visual Man-Machine Interface for Program Design and Production", in *IEEE 1986 Workshop on VIsual Languages*, 1986, pgs 71–79.

A visual interface for program design and production is discussed with regard to three aspects: (1) program design expression, (2) tool operation and (3) icon system.

As a visual interface basic scheme for (1) to (3), three factors (semantic, syntactic, and pragmatic) and policy of seven visual elements (shape, size, etc.,) are first discussed. After that, clear reasons are given for each proposal for (1) and (2). For aspect (3), an icon system is proposed for uniformly using icons with a set of tools. Practical experiments reveal that this icon system has good semantics and syntactics, but that it also has some problems.

[McWhirter93] J.D. McWhirter, Z. K. F. Eckert, and G. J. Nutt, *Using Escalante to Build Visual Language Applications*, 1993.

Constructing visual language applications is a difficult task. The Escalante system facilitates the process of application construction by supporting the high level specification of a visual language and the generation of code that realizes the language within a working application. Using Escalante one an rapidly develop highly functional applications for a wide variety of visual languages with a minimal amount of manual coding. Escalante is written in C++ and runs under X Windows. This paper presents an overview of the Escalante system and a detailed set of examples that can guide the development of visual language applications using Escalante.

[Meyer88] Bertrand Meyer, **Object-Oriented Software Construction**, 1988.

This book provides an in-depth presentation of the methods and techniques of object-oriented design, based on a careful assessment of the underlying software engineering issues.

The book review both the array of techniques needed to obtain the full extend of the approach and the design of object-oriented systems, with particular emphasis on the design of effective module interfaces.

Numerous examples of reusable software components are presented covering many of the important structure of everyday programming. Several case studies and end-of-chapter exercises support these examples.

[Meyer92] Bernd Meyer, "Pictures Depicting Pictures On the Specification of Visual Languages by Visual Grammers", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 41.

Growing interest in visual languages has triggered new extended research into the specification and parsing of multi-dimensional structures. The paper discusses the need for a visual specification formalism and introduces such a technique by augmenting logic programming with picture terms which can be considered as partially specified pictures. We define how to match picture terms and how to integrate matching with the execution of logic programs. Based upon this extension, picture clause grammars (PCGs) are introduced. PCGs are formal visual specifications of visual languages and can be used for parsing and syntax directed translation of visual languages like DCGs are used in the case of textual languages. The executability of PCGs in demonstrated by defining their translation to logic programs employing picture terms.

[Milner78] Robin Milner, "A Theory of Type Polymorphism in Programming", in *Journal of Computer and System Sciences*, 1978, pgs 348–375.

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm W which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong" and a Syntactic Soundness Theorem states that if W accepts a program then it is well typed. We also discuss extending these results to richer languages: a type-checking algorithm based on W is in fact already implemented and working, for the metalanguage ML in the Edingburgh LCF system.

[Minas93] M. Minas and G.Viehstaedt , "Specifications of Diagram Editors Providing Layout Adjustments with Minimal Change", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 324–329.

Editing diagrams conveniently requires edit operations and automatic layout tailored to the type of diagram. This necessitates a syntax-directed editor for diagrams, called diagram editor herein. We describe the basics of a generator for interactive diagram editors that offers a number of significant advantages over previous approaches. The foundation is an ew incremental algorithm for constraint evalutaion. Constraints can be specified not only by equations, as in earlier work, but also by linear inequalities. This opens the door to integrating automatic diagram layout with user-defined modifications. Furthermore, the algorithm ensures that layout adjustments initiated by user action are confined to the smallest possible part of the diagram around the point of modification, thus realizing a principle of minimal change.

[Monden84] N. Monden, T. Yoshino, M. Hirakawa, M. Tanaka, and T. Ichikawa , "HI-VISUAL : A Language Supporting Visual Interaction in Programming", in *IEEE 1984 Workshop on Visual Languages*, 1984, pgs 199–205.

Visual icons can represent the objects of a system and, at the same time, the functions which they perform. Thus the visual icon works as a tool for specifying system functions, and makes it easier to develop the system itself. Furthermore, the system thus developed can also be activated by the use of visual icons.

In order to offer an environment which makes feasible the dvelopment of a system by the use of visual icons, it is necessary to provide a software tool which supports generation and interpretation of visual icons, and organization nad evaluation of icon-based system performance. This can be regarded as a type of programming language.

This paper presents a languaged named HI-VISUAL which supports visual interaction in programming. Following a brief description of the language concept, the icon semantics and language primitives characterizing HI-VISUAL are extensively discussed. HI-VISUAL also shows a system extendability providing the possibility of organizing a high level application system as an integration of several existing subsystems, and will serve to developing systems in various fields of applications supporting simple and efficient interactions between programmer and computer.

[Myers88] Brad A. Myers , "Automatic Data Visualization for Novice Pascal Programmers", in *1988 IEEE Workshop on Visual Languages*, 1988, pgs 192–198.

Previous work has demonstrated that presenting the data structures from programs in a graphical manner can significantly help programmers understand and debug their programs. In most previous systems, however, the graphical displays, called data visualizations, had to be laboriously hand created. The Amethyst system, which runs on Apple Macintosh computers, provides attractive and appropriate default displays for data structures. The default displays include the appropriate forms for literals of the simple types inside type-specific shapes, and stacked boxes for records and arrays. In the near future, we plan to developrules for layout of simply dynamic data structures (like linked lists and binary trees) and simple mechanisms for creating customized displays. The visualizations are integrated into an advanced programming environment which is used to teach programming methodology at the introductory level.

[Najork90] Marc. A. Najork and Eric Golin, "Enhancing Show-and-Tell with a polymorphic type sysem and higher-order functions", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 215–220.

We describe enhancements to the visual dataflow language Show-and-Tell (STL). These enhancements enrich STL by a polymorphic type system similar to the one used in ML, and they introduce user-definable higher-order functions.

[Najork92] Marc. A. Najork and Simon M. Kaplan, "A prototype Implementation of the cube Language", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 270.

CUBE is a three-dimensional, visual, statically typed, inherently concurrent, higher-order logic programming language, aimed towards a virtual reality based programming environment. This paper describes a prototype implementation of CUBE.

[Najork94a] Marc-Alexander Najork , **Programming in Three Dimensions**, 1994.

This thesis describes Cube, the first visual language to employ a three-dimensional syntax. The third dimension provides for a richer syntax, makes the language more expressive, and prepares the ground for novel, virtual-reality-based programming environments. We use dimensional extent to convey semantic meaning, or more precisely, to distinguish between logical disjunctions and conjunctions and between sum and product types.

Cube uses the data flow metaphor as an intuitive way to describe logic programs. The semantics of the language is based on a higher-order form of Horn logic. Predicates are viewed as a special kind of terms, and are treated as first-class values. In particular, they can be passed as arguments to other predicates, which allows us to define higher-order predicates.

Cube has a static polymorphic type system, and uses the Hindley-Milner algorithm to preform type inference. Well-typed programs are guaranteed to be type-safe.

We have implemented two Cube interpreters: An initial feasibility study, and a prototype implementation with improved interactive capabilities. Both of them exploit the implicit parallelism of the language by simulated concurrency, implemented via time-slicing.

[Najork94b] Marc. A. Najork and Marc H. Brown, *A Library for Visualizing Combinatorial Structures*, 1994.

This paper describes Anim3D, a 3D animation library targeted at visualizing combinatorial structures. In particular, we are interested in algorithm animation. Constructing a new view for an algorithm typically takes dozens of design iterations, and can be very time-consuming. Our library eases the programmer's burden by providing high-level constructs for performing animations, and by offering an interpretive environment that eliminates the need for recompilations. This paper also illustrates Anim3D's expressiveness by developing a 3D animation of Dijkstra's shortest path algorithm in just 70 lines of code.

[Nakato] Ikuo Nakata and Masami Hagiya, **Software Science and Engineering, Selected Papers from the Kyoto Symposia**, 1991.

[Narayanan94] A. Narayanan, L. Ford, D. Manual, D. Talis, and M. Yazdani, "Language Animation", in *Proceedings of AAAI '94, Seattle, USA*, 1994.

This paper describes an intergration strategy based on designing and implementing dynamic visual primitives for language primitives. This results in an animated representation of sentences, where gaps between primitive-based language representations are bridged by visual processes. Also, language primitives which themselves represent dynamic processes (eg entering a building) can be dynamically visualized. The intergration strategy therefore provides a more powerful way of filling in gaps in primitive-based and object-centred representations than would be possible with text alone.

[Noik93] Emanuel G. Noik, "Layout-independent Fisheye Views of Nested Graphs", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 336–341.

Although a graph can be a useful device for visualizing complex relationships, drawings of large graphs can be difficult to comprehened. As one remedy, we formulated a novel generalized approach for generating fisheye views of nested graphs with multiple variable focal points, and devised an algorithm that creates fisheye views int he absence of application specific distance metrics. Previous solutions produced fisheye views by filtering or distorting drawings of graphs. Since these approaches relied on geometric notions of distance, they could only be applied effectively in limited cases. By contrast, our approach treats fisheye view generation as a phase that precedes graph layout, rather than as a technique that alters an existing drawing, and does not suffer these drawbacks.

[Nye88a] Adrian Nye, **Xlib Programming Manual for Version 11**, 1988.

This book is a complete programmer's guide to the X library, which is the lowest level of programming interface to X. It has a companion volume, Xlib Reference Manual for Version 11.

[Nye88b] Adrian Nye, **Xlib Reference Manual for Version 11**, 1988.
This book is a complete programmer's guide to the X library, which is the lowest level of programming interface to X. It has a companion volume, Xlib Programming Manual for Version 11.

[Orefice92] S. Orefice, G. Polese, M. Tucci, G. Tortora, G. Costagliola and S. K. Chang, "A 2D Interactive Parser for Iconic Languages", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 207.
In this paper we give algorithms for the construction of a 1D interactive parser which helps a user to construct correct two-dimensional iconic sentences according to positional LALR grammars. At each step of the parsing process the user is provided with a feasible set of icons. Moreover, the areas on the screen where each icon in the set may be placed are highlighted. In this way both syntactic and structural errors are avoided.

[Osterhout94] John K.Osterhout, **Tcl and the Tk Toolkit**, 1994.
The Tcl scripting language and the Tk toolkit—a programming environment for creating graphical user interfaces under X Windows—together represent one of the most exciting innovations in X Window System programming. Because Tcl and Tk are so easy to learn, extremely powerful, and contain so many sophisticated features, they have dramatically reduced development time for thousands of X programmers.

[Pandey93] Rajeev K. Pandey and Margaret M. Burnett, "Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 344–351.
Empirical studies comparing the effectiveness of visual languages versus textual languages have rarely been attempted. Here we describe an experiment conducted on programmers solving vector and matrix manipulation tasks using the visual language Forms/3, the textual language Pascal, and a textual matrix manipulation language with the capabilities of APL. Presented here are our motivation, experimental approach, some of the difficulties experienced in attempting this type of empirical study, and a summary of the experimental results and insights gained.

[Paulson91] L.C. Paulson, **ML for the Working Programmer**, 1991.
This teaches the methods of functional programming—in particular, how to program in Standard ML, a functional language recently developed at Edinburgh University. The author shows how to use such concepts as lists, trees, higher-order functions and infinite data structures and includes a chapter on formal reasoning about functional programming. The reader is assumed to have some experience in programming in conventional languages such as C or Pascal.

[Pemberton] Steven Pemberton, "Programming Aspects of Views, an Open-Architecture Application Environment", in *Proceedings of the International Workshop AVI '92*, 1992, pg 223.
Views is an open-architecture application environment that offers a consistent user interface across applications, interoperability between them, much less programming to produce an application, and the ability to add and modify applications on the fly. The system kernel contains a user interface layer and a persistent data-storage layer, so that applications only have to implement the true functionality. Objects contain no information on how they are to be displayed, so the presentation of documents can be changed easily, even on the fly. The main implementation technique is two-way invariants between objects. This document describes some aspects of programming for Views.

[Pfeiffer90] Joseph J. Pfeiffer, Jr. , "Using Graph Grammers for Data Structures Manipulation", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 42–47.

Programming language advances have generally come from replacing abstractions based on the machine with abstractions based on the algorithm. Examples include FORTRAN expressions replacing explicit reference to memory locations and registers; Algol language structures replacing explicit GOTO statements in branching; and LISP generalized lists replacing pointers in data structure manipulation. In this paper, we discuss the replacement of pointers with graph grammar productions. Such a replacaement provides a substantial improvement in the programming model used; makes better use of current high-resolution screen technology than a strictly text-based language; and provides improved support for parallel processing due to characteristics of the graph grammar formulation used.

[Pfeiffer92] Joseph J. Pfeiffer, Jr. , "Parsing Graphs Representing Two Dimensional Figures", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 200.

Generalized Two Dimensional Context Free Grammars, an extension of context free grammars to two dimensions, is described. This extension is a generalization of Tomita's Two Dimensional Context Free Grammars, and better fits into the families of graph grammars described by Crimi (Relation Grammars) and by Flasinski (edNLC Grammars). Figure Grammars are particularly useful for applications such as hand-written mathematical expressions. A two dimensional extension of the Cocke-Kasami-Younger parser for context-free languages is used to parse figures using these grammars.

[Pong86] Man-Chi Pong, "A GRAPHICAL LANGUAGE FOR CONCURRENT PROGRAMMING", in *IEEE 1986 Workshop on VIsual Languages*, 1986, pgs 26–33.

This paper presents the rationale of employing interactive graphics to support concurrent programming and describes the graphical programming language Pigsty designed for the purpose. Pigsty is based on Pascal and CSP. It uses a graphical form to represent the structure of a system of processes. I-PIGS is the programming environment designed to support Pigsty. It can execute the graphical representation of a concurrent program directly via a simulated concurrent execution mechanism. It animates the data communication between the processes in one of the windows of the modern workstation on which it runs. Other windows are used to show the structured chart form of control constructs of a process and tables of variables used in a process. The animations of data communication, logic flow and change of variable values allow the programmer to understand the dynamic behavior of a concurrent program more clearly and locate errors more easily. I-PIGS also detects deadlock during execution of a concurrent program. These capabilities of I-PIGS can help the programmer to develop concurrent programs.

[Poswig92] Jörg Poswig, Klaus Teves, Guido Vrankar, and Claudio Moraga, "VisaVis - Contributions to Practice and Theory of Highly Interactive Visual Languages", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 155.

Higher-orderness, highly interactive ,a great amount of flexibility, color, increasing the visual extent and parallelism are all of them catchwords related to the development of visual languages being in the focus of attention in the recent years. The paper reports on the functional visual language VisaVis coming up with a new user interaction strategy integrating higher order functions smoothly. VisaVis uses colors and shadows to convey information about the degree of interaction. A comparison with existing visual languages is presented to picture reached results to the reader. The translation into a metalanguage which preserves inherent parallelism in a visual program is outlined. Additionally the concept of an implicit type system is introduced being sound and complete which prepares for prevention of run-time errors and increases the visual extent at the same time.

[Poswig93a] Jörg Poswig and Claudio Moraga, "Incremental Type Systems and Implicit Parametric Overloading in Visual Languages", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 126–133.

A primary goal of much visual language research is ultimately to further the ability of visual languages to be used for realistic programming projects. As a step in this direction one expects much of incremental type systems in order to prevent run-time errors as early as possible and to preserve the user's conceptual model of a visual language at the same time. The paper reports on the integration of both an incremental type system and the support of user-definable overloaded functions in an implicit manner for the higher order visual language VisaVis. As a consequence the idea of parametric polymorphism used in many approaches for type systems is not sufficient in our approach. The concept is based upon a generalization of definite databases leading to a PROLOG interface being a main part of the type checker. Beside this data structures are discussed performing the required unification process as well as the preparation of queries for the database.

[Poswig93b] Jörg Poswig, Guido Vrankar, and Claudio Moraga, "Interactive Animation of Visual Program Execution", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 180–187.

Software visualization refers to graphical views on computer programs in general. There are only few attempts towards animation systems of visual program execution. Since programming languages nowadays are more or less complete software development environments such tools are extremely desirable also in the area of visual languages. The paper reports on the integration of such an animation tool into the visual language VisaVis taking possible parallel evaluations into account. In order to support abstractness the user can skip possibly less interesting parts of a visual program during the animation.

[Poswig94] Jörg Poswig, Guido Vrankar, and Claudio Morara, "VisaVis: A Higher-Order Functional Visual Programming Language", in *Journal of Visual Languages and Computing*, 1994, pgs 83–111.

The paper presents the functional visual language VisaVis. We focus on the new, flexible interaction strategy substitution, that brings ease of construction to visual programs and integrates higher-order functions smoothly. In order to illustrate the capabilities of the implemented prototype, comparisons with visual languages are given throughout the text. The programming environment is outlined as well as the compilation into the meta-language FFP preserving (data-) parallelism inherent in the visual programs. Improvements to the programming environment are discussed.

[Preece94] Jenny Preece, **Human-Computer Interaction**, 1994.

Offering a highly comprehensive account of the multidisciplinary field of HCI, this book illustrates the powerful benefits of a user-oriented approach to the design of modern computer systems. It discusses the technical and cognitive issues required for understanding the subtle interplay between people and computers, particularly in emerging fields like multimedia, virtual environments and computer supported cooperative work.

[Raeder85] Georg Raeder, "A Survey of Current Graphical Programming Techniques", in *IEEE Computer*, 1985, pgs 11–25.

Representing programs in graphical images offers many opportunities for improving programming techniques. This article exmaines the use of pictures in programming and compares various graphical programming systems. Topics include: pictures vs. text, use of pictures in programming, CAD/CAM, related fields.

[Ramalingam94] G. Ramalingam and Thomas Reps, "An Incremental Algorithm for Maintaining the Dominator Tree of a Reducible Flowgraph", in *Conf. Record of the Annual ACM Symposium on Principles of Programming Languages 1995*, 1994, pgs 287–296.

We present a new incremental algorithm for the problem of maintaining the dominator tree of a reducible flowgraph as the flowgraph undergoes changes such as the insertion and deletion of edges. Such an algorithm has applications in incremental dataflow analysis and incremental compilation.

[Reiss93] Steven P. Reiss, "A Framework for Abstract 3D Visualization", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 108–115.

This paper describes a package, PLUM, we have developed for visualizing abstract data in three dimensions. We are particularly interested in visualizing information about programs, both static and dynamic, but the package should have a more general applicability. The package provides a framework to support a wide variety of different 3D visualization techniques, many of which have been implemented. The package also provides support for 3D graph layout using a variety of different layout heuristics.

[Roberts88] Jim Roberts, John Pane, and Mark Stehlik, "The Design View: A Design Oriented, High-Level Visual Programming Environment", in *1988 IEEE Workshop on Visual Languages*, 1988, pgs 213–220.

Top-down design is an accepted technique for program development and is an integral part of the Introductory Computing courses at Carnegie Mellon University. This planning process works well in the abstract, but novices typically abandon this technique as soon as implementation begins, because traditional methods of program construction tend to focus immediate attention on low-level details. This paper proposes a concise graphical convention for representing a problem decomposition that can be used on paper, in the classroom, and on the computer. It then proposes an implementation of this convention, as an extension to an existing structure-editor programming environment, that allows high-level design to take place online. As the user graphically edits this design, the system silently tracks the set of low-level details that are necessary to ultimately conform the program code with the design.

The necessary changes are then presented sequentially to the user after the design phase is complete. It is further suggested that this convention is useful as a way to view already-completed programs, adn as a tool for debugging.

[Robertson91] Scott P. Robertson, Gary M. Olson, and Judith S. Olson, *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991.

CHI '91 Conference Proceedings Human Factors in Computing Systems, April 27-May 2, 1991, New Orleans, Louisiana, USA

[Robertson91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card, "Cone Trees: Animated 3D Visualizations of Hierarchical Information", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 189.

The task of managing and accessing large information spaces is a problem in large scale cognition. Emerging technologies for 3D visualization and interactive animation offer potential solutions to this problem, especially when the structure of the information can be visualized. We describe one of these Information Visualization techniques, called the Cone Tree, which is ued for visualizing hierarchical information structures. The hierarchy is presented in 3D to maximize effective use of available screen space and enable visualization of the whole structure. Interactive animation is used to shift some of the user's cognitive load to the human perceptual system.

[Rogers90] Greg Rogers , "The GRClass Visual Programming System*", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 48–53.

Visual programming techniques have proven successful within limited domains. However, little progress has been made in using graphics to support "real-world" programming. The GRClass system attempts a solution by combining Graphics, Relations, and Classes to provide a visual interface for programming graph data structures within an object-oriented framework. This is done by extending the object-oriented model with inter-object relations. These relations are then used to directly implement the conceptual model of the graph data structures. Within the GRClass framework, data structures are objects that maintain relation tables. These relations and the objects participating in the relations constitute the form of the data structure. A graphical notation is used to specify the possible relations and to mainpulate the relation graph. GRClass is implemented within the Andrew Toolkit programming environment.

[Roseler91] Randy Roseler, "Position Paper", in *Addendum to the Proceedings OOPSLA 91*, 1991, pg 105.

[Rosson91a] Mary Beth Rosson, John M. Carroll, and Christine Sweeney, "A View Matcher for Reusing SmallTalk Classes", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 277.

A prime attraction of object-oriented programming languages is the possibility of reusing code. We examine the support provided by Smalltalk to programmers attempting to incorporate an existing class into a new design, focussing on issues of usage examples, object-oriented analysis, how-to-use-it information and object connections. We then describe a View Matcher for reuse, a tool that documents reusable classes through a set of coordinated views onto concrete usage examples; in three scenarios, we illustrate how the tool addresses the issues raised in our analysis of reuse in Smalltalk.

[Rosson91b] Mary Beth Rosson, John M. Carroll, and Christine Sweeney, "Demonstrating a View Matcher for Reusing SmallTalk Classes", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 431.

no abstract provided

[Rush95] Gary Rush, *Draft Document describing Newspeak*, 1995.
Introduction :

Newspeak is an object-oriented visual programming language that is inherently parallel and general purpose. As an object-oriented language it contains the traditional concept of objects containing a set of attributers and having a defined interface. The object-oriented paradigm is extended to include a concrete relationship between library and class, virtual classes and objects, callbacks and a powerful template mechanism.

The language is general-purpose in that it only represents a mechanism for describing a system, and does not include any domain-specific features. In all cases, the goal of the language is to provide a means of systematic extension rather than building in features. This concept goes beyond existing extendible language to the extent that Newspeak is an abstract programming language. The language is not bound to the machine-executable environment but is also capable of representing shell scripts, files, networks and any other system that can be described in terms of class and function.

Newspeak is an inherently parallel language in that programs are written in the same way regardless of whether the target machine has multiple processors or threads. A program is written in terms of those portions that can be executed in parallel and in sequence, and it is the responsibility of the compiler to generate code appropriate to the target platform. The language represents static parallel relationships in terms of threads, and dynamic relationships with functions.

Finally there is the visual nature of the language. Newspeak has no textual form either a a user interface or as an underlying representation. A development environment for the language represents classes, objects and functions as icons. In a function description, objects are passed to functions called of other objects in a horizontal control flow. Concurrent sections of code are shown as vertically stacked lines, running in parallel with code.

[Sammet69] Jean E. Sammet, **Programming Languages: History and Fundamentals**, 1969.
This is a good introduction to the early history of programming languages (1969).

[Sargent93] Robert G. Sargent and Douglas G. Fritz, *Hierarchical Control Flow Graph Models*, 1993.

A general paradigm using two complementary types of hierarchical model specification structures is developed for the specification of discrete event simulation models based on encapsulated atomic model components that communicate solely via message passing. One type of structure specifies the hierarchical coupling of model components and how the atomic components of the model can interact. The other type of structure specifies the internal behavior of each atomic component in a hierarchical way. The model elements of both specifications are encapsulated and reusable. Hierarchical Control Flow Graph Models, a special case of this general paradigm, is developed as a hierarchical extension of Cota and Sargent's Control Flow Graph Model representation. Both a coupling extension and a hierarchical behavior extension are developed.The coupling extension adds support for coupled model components and the behavior extension introduces the concept of a Macro Control State which encapsulates a partial behavior specification for an atomic component. A systems theoretic representation for the atomic components of Hierarchical Control Flow Graph Models is given. Algorithms are presented for transforming Hierarchical Control Flow Graph Models into equivalent nonhierarchical Control Flow Graph Models which can then be executed using existing algorithms for Control Flow Graph Models. Lastly, two of the sequential algorithms for the execution of Control Flow Graph Models are presented.

[Self95] "The Self Group", *Self 4.0 Read This First*, 1995.

This is a description of the Self project, its history and present state, notes on the 4.0 release, documentation descriptions and installation instructions.

[Selker88] Ted Selker and Larry Koved , "Elements of Visual Languages", in *1988 IEEE Workshop on Visual Languages*, 1988, pgs 38–44.

Visual language is the systematic use of visual presentation (graphic) techniques to convey meaning. This paper proposed a structural classification and vocabulary for these languages. Visual languages, like verbal languages, are defined by a grammar and semantics. This paper defines the visual grammar, the elements of visual language.

Visual elements are composed of:

visual alphabet	a set of visual primitives used in visual language
visual syntax	composition of primitives to form visual statements
interaction	user to system communication
structure	rules combining sublanguages into a language

The classification of the visual elements can be viewed as a linguistic description of visual language.

[Shilling92] John J. Shilling and John T. Stasko, *Using Animation to Design, Document and Trace Object-Oriented Systems*, 92.

Current diagraming techniques for the dev. and documentation of object-oriented designs largely emphasize capturing relationships among classes. Such techniques cannot capture full designs because the static nature of cls relationships makes them inadequate for describing the dynamics of object collaboration. Other techniques attempt to diagram dynamic behavior but are limited by their media to producing essentially passive description of dynamic operations. What is still needed is a technique that models message ordering, changing visibility, and temporal object lifetimes in a manner that is concise and immediate. We have developed an approach in which developers use animation to develop and capture object-oriented designs. This allows developers to design object-oriented scenarios in the way that they visualize them: by animating the actions of the objects in the scenario. The same animation then acts as the documentation for the design. Its playback makes immediately evident the temporal relationship of object messages, object creation, object destruction, and changing object visibility. Our technique is supported as part of a suite of object-oriented development tools we call GROOVE.

[Shu] Nan C. Shu, "FORMAL: A Forms-Oriented, Visual-Directed Application Development System", in *IEEE Computer*, 1985, pgs 38–49.

What you sketch is what you get. A forms-oriented programming language simplifies the process of representing both data objects and program structure. Nonprogrammers can easily use such a language to create and computerize their own application programs. Paper addresses and describes FORMAL, Forms-Oriented Manipulation Language. Includes diagrams.

[Simmel91a] Sergiu S. Simmel , "Object-based Visual Programming Languages", in *Addendum to the Proceedings OOPSLA 91*, 1991, pgs 99 – 102.

The guiding questions posed for a Birds of a Feather Session at OOPSLA 91 on Object-based Visual Languages were these (summarized):

- what do you expect the visual aspect of OBVPLs buy you and your customers?
- are the benefits of an OBVPL actual results of a graphical visualization, or did the constraints of such a visualization force the language designer to keep the concepts simpler, higher level, and generally cleaner?
- what is the "proper" granularity of objects in OBVPLs?
- is a class-based model appropriate for a VL?
- assuming that OBVPL means that the PL is OB and not just implemented that way, where are the objects?
- should OBVPLs be general purpose, or are the visual means at their best in the context of a more specialized language?
- should OBVPLs be designed with few but high combinable primitives or should they have larger number of low and high level, convenient facilities?
- how complex should the visual syntax be?
- when is a dataflow model appropriate for an OBVPL? is control flow more appropriate? is a communications model more appropriate?
- what examples of lexical and syntactic elements in OBVPLs have been found particularly useful and versatile?
- which parts of the application development are easier/harder with OBVPLs than with textual OBLs?
- how do various application domains influence or determine the answers to all of these?

[Simmel91b] Sergiu S. Simmel , "Two Observations on Visual Languages", in *Addendum to the Proceedings OOPSLA 91*, 1991, pgs 103 – 105.

[Singley91] Mark K. Singley , "Molehill: An Instructional System for Smalltalk Programming", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 439.

no abstract provided.

Summarizes Smalltalk as a good platform for rapid prototyping and software reuse that is generally regarded as difficult to learn. Molehill is an instructional system for Smalltalk which combines elements of intelligent tutoring technology with more tool-like elements like hypertext and keyword search. The intended audience for the language is experienced procedural programmers. Specific instructional goals include:

• support of code comprehension and browsing as well as code generation...

• manage the transfer from procedural programming to object-oriented programming...

• guide students' actions and deliver help through the reification of goals and plans.

[Slack92] Jon Slack and Cristina Conati, "Encoding Information through Spacial Relations", in *Proceedings of the International Workshop AVI '92*, 1992, pg 85.

Advanced visual interfaces need to exploit the advantages of encoding information through spatial relations. Spatially encoded information is highly accessible and the spatial medium affords simple implementations of cognitive operations that are expensive to compute symbolically. These advantages accrue from the rapid perceptual operations that identify and encode the spatial relations implicit in the visual array. The paper outlines a representation framework for the cognitive/perceptual encoding of graphically presented information. The processes that access the information by extracting and decoding spatial relations can be specified within this framework. The work also provides a basis for costing the information extraction routines, enabling the notion of "effective graphics" to be quantified. Such costings are particularly useful in the context of planning multimedia presentations of information. A detailed worked example shows how these ideas can be applied to the generation of optimal graphical formats for quantitative relational data.

[Smith86] Randall B. Smith, "THE ALTERNATE REALITY KIT. An Animated Environment for Creating Interactive Simulations", in *IEEE 1986 Workshop on VIsual Languages*, 1986, pgs 99–106.

The Alternate Reality Kit (ARK) is an animated environment for creating interactive simulations. ARK simulations are intended to enable the development of intuitive understanding of fundamental simulation laws by making these laws appear as accessible physical objects called interactors^. Although general programming can be done in ARK, this paper will concentrate on describing the metaphor and basic functionality of the environment. Several of the more important ARK objects are described, including the mouse-operated hand icon which is the user's means of interacting with the system. Examples of a user playing in a planetary orbit simulation demonstrate how buttons and message boxes are used to communicate with ARK objects. An ARK environment may consist of several windows, each containing its own distinct simulated world called an alternate reality. A simulated three-dimensional domain called meta reality surrounds these alternate realities. Meta reality is used for representing actions "beyond" the two-dimensional alternate realities. The ARK interface is built upon a physical metaphor in which every object has position and velocity, and can experience forces. The intention is to create an easily understood interface by exploiting the user's existing expertise in dealing with everyday objects.

[Smith88] R.B. Smith, "Experience with the Alternate Reality Kit - an example of the tension between literalism and magic", in *Mini and Micro computers and their applications*, 1988, pgs 1 – 8.

[Spratt93] Lindsey Spratt and Allan Ambler, "A VISUAL LOGIC PROGRAMMING LANGUAGE BASED ON SETS aND PARTITIONING CONSTRAINTS.", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 204–208.

This paper presents a new programming language named SPARCL that has four major elements: it is a visual language, it is a logic programming language, it relies on sets to organize data, and it supports partitioning constraints on the contents of sets. It is a visual programming language in that the representation of the language depneds extensively on non-textual graphics and the programming process relies on graphical manipulation of this representation. It is a logic programming language in that the underlying semantics of thel anguage is the resolution of clauses of a Horn-like subset of first order predicate logic. It uses sets as the only method of combining terms to build complex terms. Finally, one may constrain a set"s structure by specifying a partitioning into pairwise disjoint subsets.

[Stasko90] John T. Stasko, "Simplifying Algorithm Animation with TANGO", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 1–6.

Algorithm animation is the process of abstracting the data, operations, and semantics of computer programs, and then creating dynamic graphical views of those abstractions. Dynamic visualizations of algorithms are useful for understanding programs, evaluating programs, and developing new programs. Existing algorithm animation systems lack formality, thus making the algorithm animation process difficult, inconsistent, and inflexible. We develop a conceptual framework for algorithm animation to modularize and simplify the animation design process. The framework introduces the path-transition paradigm, a model consisting of abstract data types appropriate to create smooth, continuous image movement.

Concurrently, we have developed an algorithm animation system called TANGO (Transition-based ANimation GeneratiOn), implemented based on the framework. TANGO provides animation designers with the capabilities to produce sophisticated, real-time, two-dimensional color views of programs without low-level graphics coding. TANGO supports a clean separation between programs and animations, resulting in a flexibility to map one or several programs to more than one animation view—a feature useful not only for experimenting with many views of a simple program, but also for more sophisticated animations such as parallel process visualization.

[Stasko91] John T. Stasko, "Using Direct Manipulation to Build Algorithm Animations by Demonstration", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 307.

Dance is a tool that facilitates direct manipulation, demonstration development of animations for the Tango algorithm animation system. Designers sketch out target actions in a graphical-editing fashion, then Dance automatically generates the code that will carry out those actions. Dance automatically generates the code that will carry out those actions. Dance promotes ease-of-design, rapid prototyping, and increased experimentation. It also introduces a methodology that could be used to incoporate demonstrational animation design into areas such as computer assisted instruction and user interface development.

[Stasko92] John T. Stasko and Charles Patterson , "Understanding and Characterizing Software Visualization Systems", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 3.

The general term software visualization refers to graphical views or illustrations of the entities and characteristics of computer programs and algorithms. This term along with many others including data structure display, program animation, algorithm animation, etc., have been used inconsistently in the literature, which has led to confusion in describing systems providing these capabilities. In this paper we present a scaled characterization of software visualization terms along aspect, abstractness, animation, and automation dimensions. Rather than placing existing systems into hard-and-fast categories, we focus on unique and differentiating aspects across all systems.

[Stasko93] John T. Stasko and Joseph F. Wehrli, "Three Dimensional Computation Visualization", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 100–107.

Systems supporting the visualization and animation of algorithms, programs, and computations have focused primarily on two-dimensional graphics to date. In this paper we identify the benefits and the drawbacks of using three-dimensional graphics in these types of systems, and we describe how 3D imagery can be used for visualizing computations in interesting new ways. We also present examples of 3D computation visualizations created with a new toolkit that we have developed. The toolkit has been extended to run in a virtual environment and we describe our early interactions with it.

[Steele84] Guy L. Steele, Jr., **Common LISP**, 1984.

This describes the first version of Lisp designed to be used as a common dialect by all workers in the field of artificial intelligence. Common Lisp is the culmination of three years of collaborative work by more than 60 contributors from government, industry, and academia. The book includes: language specifications, descriptions of all standard language constructs, notes on key differences between Common Lisp and other dialects, and notes on ways to implement unique or ambiguous cases.

[Steele90] Guy L. Steele, Jr., **Reference for Common Lisp**, 1990.

[Stiles92] Randy Stiles and Michael Pontecorvo, "Lingua Graphica: A Visual Language for Virtual Environments", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 225.

This paper describes work in progress on a visual programming language for virtual environments, called Lingua Graphica. Three-dimensional, solid language constructs are used to visualize and compose programs and higher-level scripts while inside of a virtual environment. This is a departure from the normal way that virtual environments are developed and used. Using Lingua Graphica to specify the behaviors directly associated with virtual objects will decrease the time required to create virtual environments for training, design, and the communication of ideas.

[Stotts90] P. David Stotts, "Panel on Hypertext Systems", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 66–68.

This panel on hypertext systems is convened to discuss the aspects of current hypertext models and systems that are germane to visual programming. The sections below each present the position of a different research project or gruop, addressing these questions:

- why is your work both "hypertext" and "visual programming"?
- are there other overlaps you see that may not necessarily be demonstrated by your work?
- what are the distinct and defining differences between these two domains?
- what are the interesting topics for further research in the combined domain?

The panel is intended to explore the relationship between what are now thought of by most as two separate areas, but are seen by the panel members to have clear similarities. The applications of each domain to the other, and the mutually beneficial lessons that can be learned from work int he two areas, are outlined in the project descriptions following.

[Stroustrup90] Bjarne Stroustrup, **The C++ Programming Language, 2nd edition**, 1990.

Written by the designer of C++, this book is the definitive guide to C++, its fundamental principles, and key techniques for mastering it. The book provides coverage of all C++ features, including exception handling, templates (parameteried types), and the latest ANSI/ISO extensions. This should be in every C++ programmer's library.

[Sugihara93] Kazuo Sugihara, Koji Takeda, and Mitsuyuki Inaba, "An Approach to Animation of Software Specifications", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 374–375.

This paper addresses animation of software specifications that can ease understanding of the properties of software specification which are not easy to capture from the specifications alone. It is useful for program understanding, testing, simulation, impact analysis, etc., in software development and maintenance. We present basic concepts of a visual language for specifying animation of software specifications. In our approach, a scenario of animation is specified by using a Petri net, which enables us to describe concurrency and synchronization in animation, and audio-visual effects are associated with objects in software specifications. This is a step toward auto-visual specifications of a software system.

[Suleiman92] Khalid A. Suleiman and Wayne V. Citrin, *An International Visual Language*, 1992.

We present an experiment supporting the thesis that visual languages are well suited for programmers whose native language is not English. We first analyze the particular problems of our target group, Arabic speakers, in comprehending Pascal program structures, and we present a number of previously proposed solutions to this problem. We then present our own solution, a visual programming environment in which all syntactic and nearly all semantic information is presented in a BLOX-like notation. Only labels, variable names, expressions, and comments are presented textually, in Arabic. Finally, we present the results of an experiment in which the system received a high degree of acceptance among Arab speakers. The system we describe is easily retargetable to other languages, and we believe that the advantages of such a system for Arabic speakers are equally applicable to the speakers of other languages.

[Sun Microsystems89 (formerly Sun89)] Sun Microsystems, Inc., **Open Look. Graphical User Interface Functional Specification**, 1989.

This book is a comprehensive guide to the Open Look user interface. It is written for developers creating Open Look toolkits and designing user interface applications. It contains clear and consistent discussions of all features.

[Swenson93] Keith D. Swenson, "A VISUAL LANGUAGE TO DESCRIBE COLLABORATIVE WORK", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 298–303.

In order to satisfy goals for developing collaboration software, a powerful yet simple visual language has been developed for use by end users in describing plans for work activities. The approach presented is unique because the model allows for collaboration during the planning process; different people are responsible for different parts of the plan. Process plans may be modified on the fly, to allow handling of exception and other changes. Policies may be created that automatically create a plan for a user in specific situations. Plans at different levels represent the viewpoint of the people responsible for those plans. These visual representations of plans are then used directly to facilitate the coordination of those activities.

[Tanimoto92] Steven L. Tanimoto, *1992 IEEE Workshop on Visual Languages.*

1992 IEEE Workshop on Visual Languages, Sept. 15–18, Seattle, USA

[Tanimoto93] Steven L. Tanimoto, *1993 IEEE Symposium on Visual Languages*, 1993.

1993 IEEE Symposium on Visual Languages, Bergen, Norway, Aug 24-27, 1993

[Tatsukawa91] Kosuke Tatsukawa, "Graphical Toolkit Approach to User Interaction Description", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 323.

The paper proposes a new model which describes the presentation and behaviour of user interfaces. The behavior of the user interface is specified as an event flow graph consisting of components as its nodes and the paths through which events are sent as its edges. A meta-level function is introduced to describe user interfaces whose constituent components change through user interaction. The reusibility of objects is augmented by representing their presentation and behaviour as a connected subgraph of the event flow graph. User interface development systems based on this model can create the user interface under a totally visual environment.

[Tetzlaff91] Linda Tetzlaff and David R. Schwartz, "The Use of Guidelines in Interface Design", in *Reaching Through Technology, (CHI '91 Conference Proceedings) Human Factors in Computing Systems*, 1991, pg 329.

We studied the use of an evolving interface style book to evaluate the role of such guidelines in the development of style-conforming interface designs. Although the designs were judged to be generally conforming, study participants had significant difficulty in interpreting the guidelines. Our designers were manifestly task oriented and impatient with extraneous material. They depended heavily on the pictorial examples, often to the exclusion of the accompanying test. We conclude that dependency on guidelines should be minimized, and that guidelines should be developed primarily to complement toolkits and interactive examples, focussing on information intrinsically unavailable through those vehicles.

[Tognazzini92] Bruce Tognazzini, **TOG on Interface**, 1992.

This internationally recognized interface designer, Apple Employee #66, writes about the process of design, principles to design by, and offers his view of important guidelines. The book contains fact and insight, supported by years of user feedback, organized research and common sense.

[Tonouchi92] Toshio Tonouchi, Ken Nakayama, Satoshi Matsuoka, and Satoru Kawai, "Creating Visual Objects by Direct Manipulation", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 95.

Low-cost implementation of graphical user interfaces (GUIs) have relied on the widget library framework. Although conventional widgets are suitable for developing typical GUIs with predetermined interaction styles, application-specific customization of interactions is rather difficult, especially for a non-programmer. Instead, we propose a new framework whereby the GUI designers can arbitrarily compose new visual object recursively from intrinsic primitive object. The behavior of a compose object is governed by constraints extracted from the trace of operations issued to the graphic editor. A prototype system Oak base on the framework is successfully implemented. Oak allows GUI designers to compose visual objects by direct manipulation allowing non-programmers to create customized widgets of high-degree complexity.

[Traub86] Joseph F. Traub, **Annual Review of Computer Science**, 1986.

This is a collection of various works in computer science.

[Tucci92] Maurizio Tucci, Guiliana Vitiello, Gennaro Costagliola, Guiliano Pacini, and Genoveffa Tortora, "Graphs and Visual Languages for Visual Interfaces", in *Proceedings of the International Workshop AVI '92*, 1992, pg 304.

Graphical layout facilities such as diagrams and graphs are recognized as powerful tools for visual interaction. An icon-oriented visual language is based upon a set of icons which are used for pictorially representing conceptual entities and operations. For a wide applicability of visual languages, it is then necessary to consider a general syntactical model. The present paper is concerned with the comprehension of the features that a grammatical formalism for non-linear languages must have in order to match any requirement for a satisfactory parsing phase.

With the previous extension, we aim to build a visual interactive system able to support the description, analysis, modification, implementation, prototyping, testing, of complex systems by means of graphical representations.

Often graphs have been used to represent non-linear structures, and graphs play a central role in the development of diagrammatic systems for user interaction. In order for such interaction tools to be employed in diverse environments, it can be very useful a grammatical formalism able to describe the syntax of graphs and diagrams.

The formalism of Relation Grammars provides a general framework for specifying multi-dimensional structures. It supports an easy implementation of a general parsing technique for non-linear languages. The same technique can be applied to several graph grammar formalisms that can be transformed into the RG one. We show the effectiveness of such a technique by transforming two well known graph grammar formalisms, due to Ghezzi-Della Vigna and Janssens-Rozenberg, into the RG formalism.

[Ungar91] David Ungar and Randall B. Smith, *SELF: The Power of Simplicity*, 1991.

SELF is an object-oriented language for exploratory programming based on a small number of simple and concrete ideas: prototypes, slots, and behavior. Prototypes combine inheritance and instantiation to provide a framework that is simpler and more flexible than most object-oriented languages. Slots unite variables and procedures into a single construct. This permits the inheritance hierarchy to take over the function of lexical scoping in conventional languages. Finally, because SELF does not distinguish state from behavior, it narrows the gaps between ordinary objects, procedures, and closures. SELF's simplicity and expressiveness offer new insights into object-oriented computation.

[Ungar95] David Ungar, *How to Program Self 4.0*, 1995. Introduction.

The Self programming environment provides facilities for writing programs, and the transporter provides a way to save them as source files. Of all the parts of Self, the programming environment probably has the least research ambition in it. We simply needed to concentrate the innovation in other areas: language design ,compiler technology, user interface. The Self programming environment strives to meet the high standard set by Smalltalk's, but with a more concrete feel. The transporter, on the other hand, is somewhere in-between completely innovative research and dull development. It attempts to pull off a novel feat—programming live objects instead of text—and partially succeeds. Its novelty lies in its view of programs as collections of slots, not objects or classes, and its extraction of the programmer's intentions from a web of live objects.

[VeeReeth93] Frank VeeReeth and Eddy Flerackers , "Three-Dimensional Graphical Programming in Cael", in *1993 IEEE Symposium on Visual Languages*, 1993, pgs 389–391.

In the visual programming community ,many interesting graphical representations have been reported upon. Most of them have a 2D or 2.5D appearance on the screen in order to reflect the inherent multidimensionality of the programming constructs being represented. By going into a three-dimensional representation, this reflection can go a step further. With todays computers, it moreover becomes feasible to extend the dimensionality of the program (and data structure) depiction. We follow this approach by realizing 3D graphical programming techniques within CAEL, our interactive Computer Animation Environment Language. The paper elucidates the underlying concepts, architecture and 3D representations we utilize in CAEL.

[Vick84] C.R. Vick, Ph.D., and C.V. Ramamoorthy, Ph.D., **Handbook of Software Engineering**, 1984.

This book offers the information to design, implement, test, and maintain virtually any type ofsoftware. It is the first single source of its kind to cover in detail such a wide range of topics.

[Vose86] G. Vose and G. Williams, "LabView: Laboratory Virtual Instrument Engineering Workbench", in *BYTE magazine*, 1986, pgs 84 – 92.

From Byte magazine, September 1986. Editor's note: The following is a BYTE product preview. It is NOT a review. We provide an advance look at this product because we feel that it is significant. A complete review will follow in subsequent issue.

[Wall92] Larry Wall and Randal L. Schwartz, **Programming perl**, 1992.

This is the authoritative guide to the hottest new UNIX utility in years, co-authored by the creator of that utility.

Perl is a language for easily manipulating text, files and processes. Perl provides a more concise and readable way to do many jobs that were formerly accomplished (with more difficulty) by programming in the C language or one of the shells. Even though Perl is not yet a standard part of UNIX, it is likely to be available at any UNIX site.

[Wang93] Dejuan Wang and John R. Lee, "Pictorial Concepts and a Concept-Supporting Graphical System", in *Journal of Visual Languages and Computing*, 1993, pgs 179–199.

A pictorial concept specifies a class of pictures with certain common structures and properties. Pictorial concepts (or picture specifications) are useful in various applications of graphical interfaces, and specification mechanisms have been proposed to allow putting concepts together to define new concepts with further logical constraints.

We present a new approach to construction of pictorial concepts in a structured manner based on a type-theoretic framework, which provides a richer specification language with abstraction mechanisms by means of which classes of pictures with more sophisticated structures can be specified in a clear and natural way. In particular, the approach allows us to study, define and use pictorial concepts at a higher level of abstraction so that powerful and useful operations over pictorial concepts can be defined and systematically sued to form sophisticated pictorial concepts.

A concept-supporting graphical system has been designed on the basis of the theoretical framework and an experimental implementation is described. The system supports users' definition of pictorial concepts without requiring programming expertise, and realizes concept-checking. This provides valuable support to a flexible use of pictorial concepts, which is important in effective and systematic use of pictures as meaningful visual expressions.

[Watson93] Mark Watson, **Portable GUI Development with C++**, 1993.

Cross-platform GUI development.

[Wegner90] Peter Wegner, "Concepts and Paradigms of OOP", in *OOPS Messenger*, 1990, pgs 7–87.

This paper examines goals, origins and paradigms of object-oriented programming, explores language design alternatives, considers models of concurrency, reviews mathematical models to make them accessible to non-mathematical readers, and speculates on what may come after object-oriented programming, concluding that it is a robust component-based modeling paradigm. This paper expands on the OOPSLA 89 keynote talk.

[Weise94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard , "Value Dependence Graphs: Representation Without Taxation", in *Conf. Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994, pgs 297–310.

The value dependence graph (VDG) is a sparse dataflow-like representation that simplifies program analysis and transformation. It is a functional representation that represents control flow as data flow and makes explicit all machine quantities, such as stores and I/O channels. We are developing a compiler that builds a VDG representing a program, analyzes and transforms the VDG, then produces a control flow graph (CFG) [ASU86] from the optimized VDG. This framework simplifies transformations and improves upon several published results. For example, it enables more powerful code motion than [CLZ86, FOW87], eliminates as many redundancies as [AWZ88, RWZ88] (except for redundant loops), and provides important information to the code scheduler [BR91]. We exhibit a fast, one-pass method for elimination of partial redundancies that never performs redundant code motion [KRS92, DS93] and is simpler than the classical [MR79, Dha91] or SSA [RWZ88] methods. These results accrue from eliminating the CFG from the analysis/transformation phases and using demand dependences in preference to control dependencies.

[Welsh80] Jim Welsh and Michael McKeag, **Structured System Programming**, 1980.

The purpose of this book is to demonstrate the application of structured programming to the construction of system programs—in particular compilers (which are typical of many similar text-handling programs) and operating systems (which are typical of many real-time systems).

[Wexelblat81] Richard L. Wexelblat, **History of Programming Languages**, 1981.

This is another good book on the history of languages. This is from the ACM SIGPLAN History of Programming Languages Conference, June 1978. The keynote address is by Grace Hopper, and it alone should be required reading for all language students.

[Wiederhold87] Gio Wiederhold, **File Organization for the Database Design**, 1987.

An excellent source for the gory details on the file management of complex data structures.

[Williams90] Carla S. Williams and John R. Rasure , "A Visual Language of Image Processing", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 86–91.

Cantata is a graphically expressed, data flow-oriented language which provides a visual programming environment within the KHOROS system. Originally created as a research tool for image processing, the KHOROS system includes a high-level user interface specification combined with code generators, interactive development tools, and maintenance software. The system is easily tailored to other application domains because the tools of the system can modify themselves as well as the system.

The user builds a Cantata application by connecting processing nodes to form a data flow graph. Nodes are selected from an application specific library of routines creating using the KHOROS tools, and may have arbitrary granularity, from fine to large grain. Control nodes and a parser extend the functionality of the underlying data flow methodology. Visual procedures, representing a hierarchy of subgraphs, add structure to the visual language and help to manage the complexity often associated with visual programming. A dynamic execution scheduler allows the user to interactively execute the entire flowgraph, as well as individual nodes or procedures.

[Winston84] Patrick Henry Winston and Berthold Klaus Paul Horn, **LISP, 2nd edition**, 1984.

This is a guide to symbol manipulation and basic Lisp programming as well as an introduction to the use of Lisp in practice. The use of Common Lisp is a key feature of this second edition. Other new or completely revised material treats procedure abstraction, data abstraction, debugging tools, object-oriented programming, message passing, symbolic pattern matching, rule-based expert systems, and natural language interfaces.

[Winterbottom] Phil Winterbottom, *ALEF Language Reference Manual*, XX.

Alef is a concurrent programming language designed for systems programming. Exception handling, process management and synchronisation primitives are implemented by the language. Programs can be written using both shared variable and message passing paradigms. Expressions use the same syntax as C, but the type system is substantially different. The language does not provide garbage collection, so programs are expected to manage their own memory. This manual provides a bare description of the syntax and semantics of the current implementation.

[Wittenburg90] Kent Wittenburg and Louis Weitzman , "Visual Grammars and Incremental Parsing for Interface Languages", in *1990 IEEE Workshop on Visual Languages*, 1990, pgs 111–118.

In this paper we present a grammar formalism and parsing algorithm for the purposes of defining and processing visually based languages. Our work is currently set in the context of a wider effort to process input sketched on interactive tablets and worksurfaces as well as to support interface dialogues using these technologies. After outlining the particular demands that these overall goals place on our visual language component, we present a grammar formalism and an incremental parsing algorithm that uses these grammars. We then compare our approach to others in the field.

[Wittenburg92] Kent Wittenburg, "Earley-style Parsing for Relational Grammars", in *1992 IEEE Workshop on Visual Languages*, 1992, pg 192.

Predictive, Earley-style parsing for unrestricted Relational Grammars faces a number of problems not present in a context-free string grammar counterpart. Here a subclass of unrestricted Relational Grammars called Fringe Relational Grammars is proposed along with an Earley-style recognition algorithm. The grammar makes use of fringe elements (the minimal and maximal elements of partially ordered sets) in defining its productions. The parsing algorithm uses indexing methods based on fringe elements in order to take advantage of equivalence relations on parse table entries, thus avoiding redundant processing.

[Wolczko95] Mario Wolczko and Randall B. Smith, *Prototype-Based Application Construction Using SELF 4.0*, 1995.
This is a set of diagram/description pages, with bullet-type points on the left half of the page and explanatory text on the right. The text describes Self, its history, philosophy, use of objects and slots, etc.

[xxx94] No author listed, *An Introduction to GNU E*, 1994?.
Introduction

E is an extension of C++ designed for writing software systems to support persistent applications. GNU E is a new variant based on the GNU C++ compiler. This paper describes the main features of E and shows through examples how E addresses many of the problems that arise in building persistent systems. This document is an overview of the E language as implemented by the GNU E compiler. It is an abridged and modified version of (Rich92), which should be referred to for background information and design rational. All examples in this document have been updated to reflect usage in GNU E.

The original design of E was an extension of C++ v.1.2 (Stro86), which extended C (Kern78) with classes, operator overloading, type-checked function calls, and several other features. C++ v.2.0 has added multiple inheritance, and E has been ported to this version. This paper describes the use of the latest E variant, GNU E. GNU E is based on version 2 of the CNU C++ compiler.

GNU E differs from previous implementations based on the AT&T cfront compilation system. The only language extensions directly supported are the db type system and persistent data. Parametrized types in the form of class generators are no longer supported. C++ class and function templates are available and may be used for parametrized type support. Iterators are no longer supported as part of the language. Instead, a macro package is provided which supports most of the functionality of the original E language construct, albeit in a somewhat clumsier form.

[Yang94] Sherry Yang and Margaret M. Burnett, "From Concrete Forms to Generalized Abstractions through Perspective-Oriented Analysis Of Logical Relationships", in *1994 IEEE Symposium on Visual Languages*, 1994.

We believe concreteness, direct manipulation and responsiveness in a visual programming language increase its usefulness. However, these characteristics present a challenge in generalizing programs for reuse, especially when concrete examples are used as one way of achieving concreteness. In this paper, we present a technique to solve this problem by deriving generality automatically through the analysis of logical relationships among concrete program entities from the perspective of a particular computational goal. Use of this technique allows a fully general form-based program with reusable abstractions to be derived from one that was specified in terms of concrete examples and direct manipulation.

[Yeung88] Ricky Yeung , "MPL—A Graphical Programming Environment for Matrix Processing Based on Logic and Constraints", in *1988 IEEE Workshop on Visual Languages*, 1988, pgs 137–143.

The matrix is a commonly used two-dimensional data structure. On a two-dimensional display, 2-D data structures are more suitable for visualization than other linear structures such as lists. This paper describes a graphical programming environment for processing matrices, called MPL, in which matrices are integrated graphically as parts of the program. The system demonstrates that several ideas from programming language research—constraints, logic programming, and functionals—can be combined with visual programming techniques to form an efficient mixed graphical-textual notation. It also provides a new framework for expressing and prototyping matrix related algorithms.

[Yishimoto86] I. Yoshimoto, N. Monden, M. Hirakawa, M. Tanaka, and T. Ichihawa, "Interactive Iconic Programming Facility in HI-VISUAL", in *IEEE 1986 Workshop on VIsual Languages*, 1986, pgs 34–41.

Demand for the development of user-friendly interfaces grows as computers continue to be utilized in an ever wider variety of fields of application. Among the many trials being carried out, those utilizing of visual information, such as forms and icons, would appear to be essential, and the icon, especially, provides an effective means of improving interaction between the user and the computer.

We have already proposed a language, HI-VISUAL which supports visual interaction in programming. Programming is carried out simply by arranging icons on the screen. HI-VISUAL also shows a system extendability providing the possibility of organizing a high level application system as an integration of several existing subsystems.

In this paper, we present design and implementational issues of HI-VISUAL with the objective of attaining interactive iconic programming. The system features (1) icon-based programming, (2) visualization of data flow, (3) interactive programming capability, and (4) navigation for program development. Icons are managed, manipulated, and executed through seven icon descriptors: DATA, DATA TYPE, PRIMITIVE, PANEL, PROGRAM, CONTROL, and COMMAND icons.

[Yourdon89] Edward Yourdon, **Modern Structured Analysis**, 1989.
The author feels that today's more powerful technology has totally changed the focus and perspective of systems analysis. In this book, the author describes the new technology and how it can be used in the environment of structured analysis. Among others, the book discusses these features:

- state transition diagrams as a major new tool for modeling real-time systems
- updates the classical approach of modeling a current physical system
- explains a new approach for building an essential model
- includes case studies that illustrate the tools and techniques discussed throughout the book.

[Zhao92] Rui Zhao, "Gestural Interfaces for Diagram Editors", in *Proceedings of the International Workshop AVI '92*, 1992, pg 413.
Summarized, this paper comments on the visual aspect of visual programming as an aid in comprehension. User interfaces employed by visual programs tend to be menu or command selection; an alternative are gestural interfaces. The author believes that a combination of gestural with other interfaces offers the best means of interacting with the program.

[Zloof84] Moshe M. Zloof, "Classification of Visual Programming Language", in *IEEE 1984 Workshop on Visual Languages*, 1984, pgs 232–235.
In this short paper we will discuss the domain of visual programming, why it is presently popular, its classifications, and end with some examples stressing the difference between procedural and non-procedural visual programming.

[ZZ94] No author listed, *Zz Language, v. 4.0*, 1994?.
This is a programming manual. from the Introduction, "...Zz language is a general purpose incremental language. It is able to easily handle operator overloading and any kind of structured data.

According to us an incremental language is a langauge able to easily grow according to the users needs, it is suitable to develop complex compilers as well as simple comomand interpreters or desk calculator.

The user of AA starts using a simple interface that allows him to introduce new statements.

The user can specify the semantic of his statements using other Zz statements or routines written by the user itself in a conventional programming languages, like C language. We call these routines, usable from Aa, C-procedures.

Zz is able to be instructed to recognize quite general grammars; matching a grammar rule it can execute an action as written above. Thus one of the aim of Zz is to interface a set of C-procedures with a command language.

To develop a Zz application is quite easy using its user friendly environment, the user are encouraged to imprve the interfaces of their applications. Today, within the APE group, Zz is used in all the applications that require some user interface or command language.

Zz is able to add new words to its syntax like FORTH does, it is able to handle its code like Lisp does, it is able to handle syntaxes like YACC does.

For the compiler writers Zz is quite helpful. It does the job of a compiler compiler, but it handle the variable and other objects declarations maintaining a pure syntactic strong type checking. The compiler developed using Zz could be general like Ada and C. In our intentions Zz will be used to develop innovative Very High Level Language (VHLL) compilers with dynamic syntax capability.

—end of references—